KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

FUNCTIONAL PROGRAMMING ABSTRACTIONS FOR CP MODELING

Pieter WUILLE

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

December 2011

FUNCTIONAL PROGRAMMING ABSTRACTIONS FOR CP MODELING

Pieter WUILLE

Jury:
Prof. Dr. C. Vandecasteele, chair
Prof. Dr. B. Demoen, promotor
Prof. Dr. ir. T. Schrijvers, promotor
(Universiteit Gent)
Prof. Dr. ir. F. Piessens
Prof. Dr. ir. M. Bruynooghe
Prof. Dr. A. Dovier
(Università degli Studi di Udine)
Prof. Dr. M. De Cock
(Universiteit Gent)

Dissertation presented in
partial fulfillment of the
requirements for the degree
of Doctor in Engineering

December 2011

D/2011/7515/141
ISBN 978-94-6018-440-6

Abstract

The field of Constraint Programming (CP) provides problem-independent technology for solving combinatorial problems. The programmer describes his problem in a model, which is processed by a problem-independent solver to produce a solution. Solver implementations are an active topic of research, and many efficient solvers exist. Specific languages were created that provide CP modeling facilities, but these domain-specific languages (DSL) run behind on features and abstractions, compared to modern general purpose languages. On the other hand, using CP libraries in such a language requires boilerplate code, and is often far less declarative in nature.

Functional Programming (FP) is a programming paradigm that is based on the concept of functions as they are defined in mathematics, i.e., without side effects beyond their return value. Functional languages are declarative, offer very high level abstractions, and are easy to analyze. Furthermore, they can be used for general purposes.

The existing Monadic Constraint Programming (MCP) framework offers interesting high-level abstractions for CP in the state-of-the-art functional programming language Haskell. In the first part of this thesis, we study how to build upon it to create a practical, concise, and declarative modeling DSL for constraint problems.

Our language allows Finite Domain (FD) problems to be written using high-level expressions, while the underlying framework performs optimized translation to specific solver back-ends. Implemented back-ends include a proof-of-concept solver in Haskell, and bindings to the constraint solving library Gecode. For solving with low overhead, an extra mode is provided that generates C++ code for performing the solving at a later stage.

In the second part, we provide declarative modeling of search. Controlling the search heuristic employed by a constraint solver is often essential for good performance, but existing systems are either limited in possibilities or imperative in nature. Again, we choose to provide a DSL to do better. The Search Combinators language is declarative, concise and captures many advanced search heuristics. We implement an efficient code generator for these search heuristics, avoiding the pitfalls encountered.

Beknopte samenvatting

Het domein van Constraint Programming (CP) voorziet probleemonafhankelijke technologie voor het oplossen van combinatorische problemen. De programmeur beschrijft zijn probleem in een model, wat dan vervolgens verwerkt wordt door een probleemonafhankelijke oplosser ("solver"). Implementatie van dergelijke solvers is een actief onderzoeksonderwerp, en er bestaan verschillende efficiënte solvers. Er zijn domein-specifieke programmeertalen (DSL) gecreëerd die het modelleren van CP-problemen ondersteunen, maar deze lopen achter wat betreft mogelijkheden en abstracties in vergelijking met moderne talen voor algemeen gebruik. Aan de andere kant, het gebruik van CP-bibliotheken in een dergelijke taal vereist veel overtollige code, en is vaak veel minder declaratief.

Functioneel Programmeren (FP) is een programmeerparadigma dat gebaseerd is op het concept van functies zoals ze in de wiskunde gebruikt worden: zonder neveneffecten buiten hun retourneerwaarde. Functionele talen zijn declaratief, bieden abstracties van heel hoog niveau aan, en zijn eenvoudig te analyseren. Bovendien zijn ze algemeen bruikbaar.

Het bestaande Monadic Constraint Programming (MCP) raamwerk biedt interessante hoog-niveau abstracties aan voor CP in de programmeertaal Haskell. In het eerste deel van deze thesis onderzoeken we hoe daarop te bouwen om een praktische, beknopte en declaratieve DSL te creëren voor het modelleren van CP problemen.

Onze taal maakt het mogelijk om problemen met eindige domeinen (Finite Domain, FD) neer te schrijven middels hoog-niveau uitdrukkingen, terwijl het onderliggende raamwerk zorgt voor geoptimaliseerde vertaling naar verschillende onderliggende solvers. Zo is er een eenvoudige solver in Haskell zelf, en kan de solver-bibliotheek Gecode gebruikt worden. Voor het oplossen van problemen met minimale vertraging is er een extra mode voorzien die C++ code genereert, die het oplossen zelf in een later stadium doet.

In het tweede deel gaan we in op het declaratief modeleren van het doorlopen van de zoekboom ("search"). Controle over de zoekheuristiek gebruikt door een solver is vaak essentieel voor goede performantie, maar bestaande systemen zijn ofwel beperkt in mogelijkheden, ofwel imperatief van aard. Opnieuw kiezen we om dit aan te pakken met een eigen DSL. De Search Combinators taal is declaratief, beknopt en kan vele geavanceerde zoekheuristieken beschrijven. We implementeren een efficiënte codegenerator voor deze heuristieken, en vermijden de tegengekomen problemen.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

Computers are used for solving many problems that consist of finding a combination of values that satisfy a given set of conditions. Such combinatorial problems are called Constraint Satisfaction Problems (CSP). Solving them directly using ad-hoc imperative-style programs is a lot of work, and forces the programmer to be concerned with unimportant details, as he doesn't care about how the solution is found, only which solutions are valid.

Constraint Programming (CP) is a declarative paradigm specifically designed for such problems. The programmer writes a CP program, consisting of variables and the conditions (called constraints) they should satisfy, and a generic program — called the constraint solver — finds the solutions in an efficient way.

More specifically, a CP program consists of two parts: a description of the constraints for a solver (the constraint problem), and a description of the search heuristic. Although both are constructs of significantly higher level than a direct implementation in an imperative language, they still require attention to the irrelevant; for example:

- Writing efficient models requires knowledge of the constraint solving mechanism, and the solver's supported constraints.
- Small changes to the problem's representation often require significant changes to the constraints and variables — something that often happens during the incremental process of writing an effective CP program.
- Often several constraint problems share some problem structures. Without abstraction mechanisms, these structures need to be duplicated each time.

To deal with these concerns, *modeling* systems exist that integrate with constraint solving. We can subdivide them in two classes:

- Modeling languages: stand-alone languages whose programs are mapped to one or more constraint solver back-ends. They are usually highly declarative and allow convenient model transformations. Because designing and developing a new (modeling) language from scratch is a significant effort, existing modeling languages often run behind on the state of the art in programming language features such as static typing or abstraction mechanisms. They are not always Turing-complete.[1] Examples include MiniZinc (Nethercote, Stuckey, Becket, Brand, Duck, and Tack 2007), Essense (Frisch, Grum, Jefferson, Hernndez, and Miguel 2005) and Comet (Van Hentenryck and Michel 2005).

- Programming languages with an API for writing and solving constraint models that is tied to a particular constraint solver. This API can be an integral part of the language, or a set of library functions—or a combination of both. They are often Turing-complete and offer the abstraction mechanisms of the host language, but are sometimes more imperative in nature. More often, a single model cannot be reused with different constraint solvers nor can it be transformed. One prominent example is Gecode (Gecode Team 2006).

None of the existing approaches seem to cope with all our concerns however. We want the advantages of a high-level modeling language which works at a higher level than the individual constraint representation, but without losing the advantages modern programming languages have.

1.2 Approach

A first step in this direction is the Monadic Constraint Programming (Schrijvers, Stuckey, and Wadler 2009) framework in Haskell. It provides abstractions for constraint problems and solvers in Haskell. However, it is only a proof-of-concept implementation. In this thesis we study how to extend it to a practical constraint modeling system. This includes both a front-end modeling language and efficient back-ends, that avoid the decreased performance that results from modeling naively at a higher level.

1.2.1 Front-end language

In both cases — constraints and search — we choose for a DSL (Domain Specific Language) to allow modeling on a higher level. DSLs are languages designed for a specific purpose, with language features that correspond closely to the concepts from the domain they are for. Typical examples of DSLs include relational database query languages like SQL, hardware description languages like VHDL,

[1] Although this may have the advantage of guaranteeing termination.

and to some extent spreadsheets. By not being intended as general-purpose languages, DSLs can be very declarative — they do not require precise details about the actual execution, as that information is implicit in the language. For example, SQL implementations perform many optimizations on the queries they receive, and eventually use an "execution planner" to decide how to execute it in practice — all transparent to the user.

A stand-alone modeling DSL however, would suffer from the problems discussed in the previous section. We therefore turn to an *Embedded* Domain Specific Language or EDSL. Instead of being a language on its own which requires separate compilation or interpretation, it is embedded in an existing language — called the host language — and reuses many of its functions, including abstractions, standard libraries, type system and/or abilities to interact with other software. EDSLs exploit syntactic features from the host language, such as macros, overloading, custom operators, ... to implement their concept of an own language.

The purely functional language Haskell is a well-studied host language for EDSLs. Several features make it an attractive target: lazy execution, high level abstractions, an advanced type system, and custom operators. We will discuss these features in Section 3.7.3, after an introduction to functional programming languages and Haskell itself in particular.

1.2.2 Efficient back-ends

Modeling on a higher level risks performance penalties, as operational details that are critical for performance are hidden from the user. Therefore, measures must be taken to prevent increased runtime overhead or lower efficiency. As is often used in compiler design, models will be processed in several stages, each with their own representation of the problem, and opportunities for optimizations. To avoid overhead, we will provide the ability to generate efficient C++ code to perform the actual solving at a later point in time — for both the constraint model and its search heuristic.

1.3 Structure

The remainder of the text is divided in three parts: background, constraint modeling and search modeling.

Background In Part I we first give some background information.

- Chapter 2 gives an introduction to Constraint Programming.
- Chapter 3 talks about Functional Programming in general, and Haskell in particular.

Constraint Modeling Part II deals with the specification of CP problems in a high-level declarative way:

- Chapter 4 gives a short introduction to constraint modeling, and describes the basics of the Monadic Constraint Programming framework, a system that brings generic CP support to Haskell. The MCP system was introduced in (Schrijvers, Stuckey, and Wadler 2009).
- Chapter 5 describes our Finite Domain specific layer for MCP, which provides the DSL for constraint modeling and translation to actual solvers' constraints. This layer was introduced in (Wuille and Schrijvers 2009b).
- Chapter 6 elaborates on the topic of off-line solver, where no immediate solutions are produced, but rather efficient code is generated to perform the actual search for solutions at a later point in time. The ideas presented in this chapter were described in (Wuille and Schrijvers 2011).
- Chapter 7 presents some optimizations and improvements to the translation scheme, and concludes with a generic scheme for doing such translations in a flexible and efficient way. This scheme was described in (Wuille and Schrijvers 2010).
- Chapter 8 reviews the obtained modeling language by comparing it with an existing modeling system, MiniZinc.

Search Modeling Part III deals with the specification of search heuristics:

- Chapter 9 introduces the Search Combinators system, which provides a language for high-level specification of a search heuristics, and easy-to-implement semantics. It was introduced in (Schrijvers, Tack, Wuille, Samulowitz, and Stuckey 2011a).
- Chapter 10 focuses on specifics of the implementation of Search Combinators in functional languages, and how to exploit their power. They were described in (Wuille, Schrijvers, Samulowitz, Tack, and Stuckey 2011).

1.4 Bibliographical notes

Earlier work on Constraint Handling Rules (CHR), is not included in this thesis. CHR (Frühwirth 1998) is a high-level programming language extension based on guarded, multi-headed, committed-choice multiset rewrite rules. Originally designed for writing user-defined constraint solvers, CHR has matured as a powerful and elegant general purpose language. CHR is usually embedded in logic programming languages like Prolog, but implementations in imperative languages exist as

well. Our contribution is an integration of CHR with C, called CCHR (Wuille, Schrijvers, and Demoen 2007) and an overview of the implementation aspects and possible optimizations of CHR in imperative host languages, given in (Van Weert, Wuille, Schrijvers, and Demoen 2008).

One other publication not directly relating to the subject of this thesis is (Wuille and Schrijvers 2008). It explores the possibility of detecting the use of purely functional data structures (i.e., without destructive update), and replacing them with observably identical impure ones.

Part I

Background

Chapter 2

Constraint Programming

Constraint Programming (CP) in its most general form is the programming paradigm where problems are stated in the form of relations — called constraints — that must hold over a set of variables, while abstracting from the operational details of how the solution is obtained.

In this sense, it is said that:

> *Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.*
>
> - Eugene C. Freuder, Constraints, April 1997

Under this general definition, CP also includes specific solving techniques such as Linear Programming (LP) including (Mixed) Integer Programming (MIP), Boolean Satisfiability (SAT), Satisfiability Modulo Theories (SMT) and others. However, CP is typically used to refer to a specific general solving mechanism:

- An explicit representation of each variable's domain is maintained.
- In a process called constraint propagation, the variables' domains are observed, and used to shrink other domains further (pruning values from it).
- This process is repeated until a fix-point is reached, and no domains can be shrunk further.
- The propagation steps are interleaved with search steps. When a propagation fix-point is reached, and not all domains are singletons, an assumption is made, leading to either further constrained domains, or failure (no possible solutions left) — in which backtracking or a similar mechanism is used to return to the previous domains, and restart using a different assumption.

The main difference with LP, SAT, etc. is that no single solving algorithm is used. Instead, constraint-specific algorithms called propagators work independently from each other.

Constraint Programming's main strength lies in problems where variables have finite domains, with different types of constraints over them, especially global constraints. This includes combinatorial problems such as puzzles, but also complex planning/scheduling problems.

Several CP solver systems exist, including the C++ library Gecode (Gecode Team 2006), the G12 project's solvers[1], the stand-alone Minion (Gent, Jefferson, and Miguel 2006) system, and IBM's CPLEX.

2.1 Example

Sudoku is a well-known problem that can be solved easily using CP. By drawing analogies with how human beings solve such puzzles, some aspects of constraint programming can be explained.

8	7			6	9	2		
		5			3	4		1
	9				2			7
6		3		8			5	
		2	1			9		
	4		2	5		8		3
1			7				6	
4		8	6			5		
		6	3	9			2	4

Figure 2.1: Typical sudoku puzzle

In short, a sudoku puzzle is a (typically) 9×9 grid, each cell of which is either empty, or contains a number between 1 and 9. The cells themselves are grouped in 9 3×3 blocks. To solve the puzzle, all empty cells must be filled with numbers, in such a way that all numbers from 1 to 9 occur exactly once in each column, each row, and each 3×3 block.

One trivial way of solving a sudoku problem, is just trying to fill in numbers, and go back and retry when an inconsistency is reached — a process called

[1] See http://www.nicta.com.au/research/projects/constraint_programming_platform

backtracking. This is not how a human would solve things, but it isn't a good strategy for a computer either. A simple puzzle with 30 cells already filled, still has $9^{81-30} = 4.64 * 10^{48}$ possible (not necessarily valid) ways of filling it. As a human, you'd try to use reasoning to figure out cells where a certain number *has to be* placed, instantly reducing the number of possibilities by a factor 9.

In fact, humans tend to avoid backtracking as much as possible. Trying and erasing numbers is cumbersome, so we prefer to use rules to derive knowledge about the game situation, rather than simply trying things. We use this knowledge to limit possible solutions, until we don't find any rule anymore to apply. That is comparable to the propagation step in CP. In the ideal case, we end up with a solved puzzle. If not, some guesswork is necessary. When doing so, we'd choose a value that's not obviously leading to a contradiction, effectively using the inferred information about remaining possibilities. This is the search part.

The only difference between the human approach and the CP one, is the degree of propagation. Humans tend to use many different and complex rules to derive certain values or exclude possibilities. Since backtracking is relatively easy for a computer program, CP programs for solving sudoku problems typically use fewer and simpler rules, and rely on backtracking to do the rest.

2.2 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) , is a tuple $\langle X, D, C\rangle$, where X is a list of variables, D a list of associated domains, and C is a set of constraints. The domain D_i corresponding to variable X_i is also denoted $dom(X_i)$.

Each constraint C_j has an associated list of variables $var(C_j)$ it restricts, and a set $rel(C_j)$ of tuples of values that those variables are restricted to. $rel_{X_p,X_q,\ldots}(C_j)$ represents the projection of $rel(C_j)$ on the variables $(X_p, X_q, \ldots)$, namely the set of possible tuples of assignments to variables X_p, X_q, ... according to constraint C_j. If X_p does not belong to $var(C_j)$, $rel_{\ldots,X_p,\ldots}(C_j)$ contains all elements of D_p in its tuples.

A solution x to $\langle X, D, C\rangle$ is a tuple of values that is an element of $\bigcap_{c\in C} rel_X(c)$.

In the case of sudoku, this means $\langle X_{sud}, D_{sud}, C_{sud}\rangle$, where X_{sud} is a set of 81 integer variables $X_{1,1} \ldots X_{9,9}$, all with domain $D_{i,j} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. There are 27 constraints, 9 for the rows, 9 for the columns, 9 for the blocks. Each requires its set of 9 affected variables to have different values. For example, The fourth row constraint C_4 restricts variables $var(C_4) = \{X_{4,1}, X_{4,2}, \ldots, X_{4,9}\}$. The middle block constraint C_{23} applies to variables $var(C_{23}) = \{X_{r,c} | r \in \{4, 5, 6\}, c \in \{4, 5, 6\}\}$. Finally, all $rel(C_j)$ are the same set, namely the permutations of $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Constraint Optimization Problems A related type of problems are Constraint Optimization Problems (COPs). They are comparable to a CSP, but a

cost function O that maps assignments to the variables X to a score is added, resulting in $\langle X, D, C, O \rangle$. The goal is not to find just any assignment to X which satisfies all C, but to find (the) assignment(s) which both satisfy C and minimize or maximize the value of O.

2.2.1 Types of constraints

CP's propagation-based solving mechanism's greatest strength is that it supports arbitrary domains of variables (Booleans, integers, sets, intervals, ...), and arbitrary constraints over those, as long as there is way for enforcing them. Typically supported constraints over integer variables include:

- Basic equality and inequality ($=$, $\neq$, $\leqslant$, $<$) between two constraint variables or between a constraint variable and a constant number.
- Arithmetic constraints between variables and constants ($+$, $-$, $\times$, $/$, mod, ...)
- List operations like counting values in a list, summing them, requiring them to be sorted, ...
- General linear constraints ($a_1 * X_1 + a_2 * X_2 + \ldots + a_n * X_n \oplus c$, with $\oplus \in \{=, <, \leqslant\}$)
- The ALLDIFFERENT constraint, which requires every two elements in a list to be different ($\forall x_j, x_i \in X.i \neq j \Rightarrow X_i \neq X_j$)
- The SORTED constraint, which requires a list of variables to have ascending values

Constraints that apply to a larger/arbitrary amount of variables — informally, imposing some structure on them — are often called global constraints. Often (but not always) they can be decomposed into a (large) number of more elementary constraints. For example, the ALLDIFFERENT and SORTED constraints. As will be explained later, this is often a bad idea, as it decreases the level of pruning possible during constraint propagation.

2.3 Constraint Propagation

In practical implementations, constraints are typically not represented by maintaining a full enumeration of all the possible assignments to their variables, but using *propagators*, which programmatically enforce the restrictions in a much more efficient way.

Operationally, a central representation of the (remaining) domain of each variable is kept[2]. Propagators are activated one by one, inspecting their domains, and shrinking them when possible. When the domain of a variable X_k is modified, other propagators observing X_k are reactivated to check whether they can use this information to shrink the domains of any other variable further.

Another advantage of this approach is modularity: propagators for various constraints can be implemented independently, as they only interact with the variables' domains and not with each other.

Consider the simple constraint $V_1 + V_2 = V_3$ for natural numbers. It restricts (V_1, V_2, V_3) to $\{(v_1, v_2, v_3) \in \mathbb{N}^3 | v_1 + v_2 = v_3\}$. Since this set is infinite, it obviously can't be stored explicitly. Instead, a corresponding propagator is be created, that observes V_1, V_2 and V_3. For example, assume the domains of these variables have (through other means) been reduced to respectively $\{2, 3\}$, $\{5, 7\}$ and $\{10, 11\}$. A first step can be to remove 11 from V_3's domain, as it cannot be the sum of the two other variables. Once this is done, 2 and 5 can be removed from the others' domains, as those cannot result in a sum of 10 anymore — resulting in the solution $(3, 7, 10)$.

Formally, propagation corresponds to rewriting constraint problems to equivalent problems (same solutions), with smaller domains D.

2.3.1 Consistency levels

The purpose of propagators is to obtain variable domains that are *locally consistent* with the domains of other variables. Consistent means that the values remaining in variables' domains can still be combined with values in other variables' domains, without violating the constraints in the system.

Different levels of consistency have been defined in the literature, among them:

Node consistency This is the simplest form of consistency, which only requires domains not to include any values that are excluded by unary constraints. For example, given the constraint $V_i \neq 3$, the value 3 should be removed from V_i's domain to make it node-consistent. Formally, a variable V is node-consistent if its domain D_i does not contain any value v for which a unary constraint c exists where $v \notin rel_V(c)$. A problem is node-consistent when all its variables are node-consistent.

Arc consistency A variable V_p is called arc consistent with another variable V_q when for each value left in V_p's domain there is at least one value for V_q so their joint assignments does not conflict with any single binary constraint in the system. More formally, a variable V_1 is arc-consistent with another variable V_2 if

[2]Note that this not necessarily means an enumeration. For example, for numbers it is also possible to only keep a lower and upper bound.

D_1 does not contain values that do not occur as first tuple element in $rel_{V_1,V_2}(c)$ for any binary constraint c. The whole problem is arc-consistent if every variable is arc-consistent with every other.

Path consistency Path consistency is a generalization of arc consistency where tuples of variables are considered. A pair of variables (V_p, V_q) is arc consistent with another variable V_r if every assignment (v_p, v_q) to (V_p, V_q) that is already consistent with constraints between V_p and V_q, can be extended with at least one value v_r for V_r such that the assignments of (v_p, v_r) and (v_q, v_r) to (V_p, V_r) and (V_q, V_r) do not conflict with any binary constraints. This is equivalent with demanding that for each $(v_p, v_q) \in rel_{(V_p,V_q)}(c)$ all binary constraints, at least one v_r should exist such that $(v_p, v_q, v_r) \in rel_{(V_p,V_q,V_r)}(c)$ for each binary constraint c.

Generalized Arc Consistency Another extension — and a particularly interesting one — to non-binary constraints is generalized arc consistency (GAC). This requires that for every value left in a variable's domain, at least one assignment to all other variables must exist that is consistent with the constraints. Given a subset of the constraints, this is the strongest restriction of domains possible, and in the case of a single solution implies finding it. Many propagators have been described that obtain generalized arc consistency for a single constraint — even global constraints.

Even though many global constraints can be decomposed into more elementary constraints without affecting the solution space, this is typically a bad idea, precisely because of the level of consistency it enforces.

We can see a set of constraints combined and extended with fix-point semantics as a propagator on itself. However, the combination of GAC propagators for each of the decomposed constraints that constitute a global constraint, is often not GAC itself.

To illustrate, assume an ALLDIFFERENT(A, B, C, D) constraint, with $dom(A) = \{1, 2, 3, 4\}$, $dom(B) = \{2, 3, 4\}$, $dom(C) = \{3, 4\}$ and $dom(D) = \{3, 4\}$. A propagator enforcing GAC for the single constraint $A \neq B$ can only prune values as soon as either $dom(A)$ or $dom(B)$ becomes a singleton. For example, when considering $B \neq C$, there is no pruning possible: each value in the domains of B or C can be extended with a value for the other variable so that $B \neq C$ holds. As no variable has a singleton as domain (is *assigned*), no propagator for any of the decomposed $\neq$ constraints is able to perform any pruning at all.

Still, a propagator that enforces GAC for an ALLDIFFERENT constraint would remove 3 and 4 from the domains of A and B (as those values must be assigned to C or D), and 3 from the domain of A (as B now only has 3 left). This results in the pruned domains $dom(A) = \{1\}$, $dom(B) = \{2\}$, $dom(C) = \{3, 4\}$ and $dom(D) = \{3, 4\}$, leaving only two potential solutions, instead of the original $4 * 4 * 3 * 2 = 96$ combinations possible.

Bounds consistency Bounds consistency is a restricted form of consistency for integer variables, where only the bounds of the domain (lowest and highest value) are required to be consistent with other domains. In the case of integer problems with mostly inequalities and linear constraints, this is usually enough to solve the problem efficiently.

2.4 Search

Propagation takes care of the smart/fast part of solving problems: everything that can be inferred without spending too much effort, as propagators typically only enforce constraints over a small amount of variables.

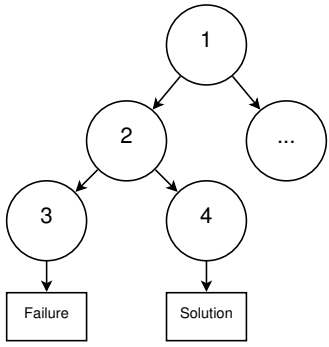

Figure 2.2: Search tree

From there on, search takes over. Search progresses by building a tree of nodes, the search tree. Each edge represents a choice being made, and each node represents a problem state. These nodes contain representations of the variables' domains and active propagators. As choices made are often incorrect, backtracking or alternative techniques (backjumping (Prosser 1993), backmarking) are used to return to a parent node in the tree, and take a different path there. Figure 2.2 shows a basic search tree. The numbers define the order in which the nodes are explored. After exploring node 3, failure occurs, and search backtracks to node 2. A different choice is made then, resulting in node 4, which gives a solution.

Search procedures — or heuristics, as they are called — range from simple algorithms that work for every constraint problem, to complex hand-optimized systems for very specific problems. In the case of default search routines, a separation between the branching (which defines the shape and structure of the tree) and the search order (which defines the order in which nodes of the tree are explored) can be made.

2.4.1 Branching

Branching describes which subnodes — with which additional constraints — are possible in given a state of the problem. Again, a wide variety of strategies is available, but typically the branching process consists of three parts:

1. Select a variable x with more than one value in its domain
2. Select a value v from this domain
3. Create subnodes with complementary constraints involving x and v

Variable selection Several variable selection strategies have been developed. As a general rule, the *first fail* principle is often followed: if one expects failure to occur, it should happen as soon as possible. Following this rule, the most-constrained variables are selected first to branch upon. In practice, an often-used rule for integers is selecting variables with the smallest domain first. Even if this does not help finding a solution, it reduces the size of the search tree: the number of leaf nodes remains equal, but the number of inner nodes decreases.

Value selection and branching constraints A separate rule is needed to select which branches to create with it. Usually either a full assignment is chosen (a $v = a$ constraint, for each value a left in $dom(v)$), an equality/inequality (a $v = a$ and a $v \neq a$ constraint, for one particular value a), or a binary split (a $v \leqslant a$ and a $v > a$ constraint). For assignment-based branchings, the values are typically chosen lowest-first, highest-first or inside-out. For binary splitting, the median is normally used.

2.4.2 Search order

The other concern when doing search, is in which order to explore the nodes of the (often implicitly) defined search tree. There are a few basic strategies.

Depth-first search (DFS) when a node is encountered that is not yet failed, but still has multiple elements left in some domains, all its branches and their descended nodes are explored immediately. It is the easiest search order to implement, as it can be implemented by simply recursively calling the search routine on each subnode. On the other hand, it can cause the search process to get stuck in a large subtree without solutions.

Breadth-first search (BFS) Only nodes on a particular distance from the root (depth) are processed at a time. When branching is needed, the subnodes are added to a queue to be processed later. When all nodes from a particular depth have been explored, processing switches to the queued nodes for a higher distance.

Iterative Deepening This is a combination of the two previous strategies: operationally, it is a depth-first search where no nodes below a certain, successively increasing, depth are explored. In a first iteration, all nodes up to depth one are explored, in a second iteration all nodes up to depth two, and so on. This effectively results in the same order of first exploration as breadth-first search, without the overhead of needing to store all successor nodes for the next depth layer. The disadvantage is that for each depth, the nodes of lower depth need to be re-explored.

Branch and bound This is a variant on DFS, specifically for optimization problems. During the search, a global variable with the lowest encountered (upper bound on) the cost function's value is kept. When a solution x is found that beats the current best, a new constraint $cost < cost_x$ is added to the problem, in order to prevent finding solutions (to the corresponding CSP) that do not improve the cost value.

Chapter 3

Functional Programming

The other paradigm used in this thesis is functional programming. In this chapter an introduction to functional programming is given.

3.1 Introduction

Functional Programming (FP) has a long history, going back to programming languages from the LISP (McCarthy 1960) family in the 60's. The main idea behind FP is focusing on using functions as they are used in mathematics: the only effect of applying a function to an argument is its return value, and no modification to state that may be kept. When this idea is followed strictly, it results in *referential transparency*, meaning that each function application in the source code can be replaced by its evaluation without changing the semantics of the program. This is in contrast with the more mainstream imperative style of programming, where execution of functions/methods/subroutines is allowed to have side effects, like changing global variables or performing I/O operations.

In practice, FP encompasses several ideas, that may or may not be considered part of FP. The availability of the following features, or the degree to which they are encouraged by the language, is typically the criterion for calling a programming language "functional":

1. Referential transparency, as explained above. Languages that enforce this principle are called *purely* functional languages.
2. Use of higher-order functions, which accept other functions as their arguments or return functions. Common routines include SORT, FILTER, MAP, REDUCE/FOLD. In functional languages, these are far more generally used.
3. *Closures* (Sussman and Steele 1998) allow one to define and create function objects within a particular scope, and still evaluate those after their defining

scope has disappeared. As this requires more than a traditional linear execution stack, this is not easily possible in traditional imperative languages like C. As a counter example, function pointers, as they are used in C, are not closures.

4. An extension to 2) and 3), *first class functions* allow one to use (unapplied) functions as values in variables, arguments and return values of other functions.

5. Algebraic data types (ADTs, see further) and pattern matching on data. Although technically unrelated to functions, this features does occur often in functional languages.

3.2 Lambda calculus

The theoretical basis for functional programming languages, is the lambda calculus, defined by Alonzo Church in the 1930's. It describes a Turing-complete system that provides a way for defining functions, and semantics for evaluating them.

3.2.1 Expressions

Lambda calculus expressions consist of only:

- *Variables*: x
- Lambda *abstractions*: if s is a lambda expression, $\lambda v \rightarrow s$[1], with v the name of a variable, is also a lambda expression. It represents an anonymous function that takes an argument that is bound to v. Occurrences of the variable v within s are called *bound* variables. For example, $\lambda x \rightarrow x$ is the identity function. $\lambda x \rightarrow 5$ is the constant function that always evaluates to 5.
- If x and y are lambda expressions, $x\ y$ represents the *application* of x on y

Brackets may be necessary to disambiguate expressions. Given that lambda abstractions define anonymous functions, a double lambda effectively defines a function that evaluates to another function when applied to a value. This is the way lambda calculus deals with multi-parameter functions, and is called *currying*. To simplify notation, the expression $f\ x\ y$ means $(f\ x)\ y$, first applying f to x, and applying the resulting function to y.

The evaluation of these expressions is defined through three term rewriting rules:

[1] Or $\lambda v.\, s$

- α renaming: changing the name of a variable being abstracted over, and every reference to it. For example, $\lambda x\ y\ z \to y\ z\ x$ could be renamed to $\lambda t\ y\ z \to y\ z\ t$.
- β reduction: the application by a lambda abstraction can be removed by substituting the bound variables with the argument of the application. For example, $(\lambda x\ y \to x\ y)\ z$ becomes $\lambda y \to z\ y$.
- η conversion: switching between $\lambda y \to z\ y$ and z.

An expression can be considered reduced to a final value when no more beta reductions are possible. Hence, in pure lambda calculus, the only type of expression consists of variables abstractions ("functions"), and every expression can be considered a function.

3.2.2 Encodings

Several encodings have been devised for more concrete data structures in lambda calculus. To give an example, the Church encoding for Boolean values:

$$
\begin{aligned}
&\mathit{true} = \lambda f\ t \to t \\
&\mathit{false} = \lambda f\ t \to f \\
&\mathit{ifthenelse} = \lambda b\ t\ f \to b\ f\ t
\end{aligned}
$$

Through the above reduction rules, we can see that:

$$
\begin{aligned}
\mathit{ifthenelse}\ \mathit{true}\ p\ q &= (\lambda b\ t\ f \to b\ f\ t)\ (\lambda f\ t \to t)\ p\ q \\
&= (\lambda b\ t\ f \to b\ f\ t)\ (\lambda x\ y \to y)\ p\ q \\
&= (\lambda t\ f \to (\lambda x\ y \to y)\ f\ t)\ p\ q \\
&= (\lambda t\ f \to (\lambda y \to y)\ t)\ p\ q \\
&= (\lambda t\ f \to t)\ p\ q \\
&= (\lambda f \to p)\ q \\
&= p
\end{aligned}
$$

Comparable schemes exist for natural numbers, and more complex data types.

3.2.3 Evaluation strategies

The reduction rules for lambda calculus only define equivalence between expressions, and do not give an evaluation strategy. Several strategies are still possible:

- *Call-by-value*: arguments to functions are always evaluated first, applying reduction rules to the innermost expressions first. This loosely corresponds to the evaluation order used by traditional languages: as the implementation only permits function arguments to be fully evaluated values, they must be evaluated before the function applied to them can be executed.

- *Call-by-name*: the outer functions are applied first, passing the unevaluated arguments to it. Those are only be evaluated when the outer function application requires them.

- *Call-by-need*: arguments to functions are passed as an unevaluated *thunk*. When the outer function requires the thunk's value, it is evaluated, and overwritten by its evaluation. This is semantically equal to call-by-name, but prevents double evaluation of the same argument. This strategy results in *lazy evaluation*, where execution follows data flow.

3.3 Type systems

As soon as other types of data constants are allowed within lambda expressions directly, such as integers, one loses the property that every expression can be evaluated. For example, arithmetic operations cannot be applied to functions, only to integers. To determine which types of arguments are allowed in applications, type systems (Pierce 2002) were developed. Types in this sense are static properties of expressions, that allow us to prove that no invalid applications will occur during evaluation.

3.3.1 Simply typed lambda calculus

One can define types for lambda expressions as being either:

- A constant type, like *Int*, referring to integer expressions.

- A function type $\alpha \to \beta$, where α and β are types themselves.

However, if we require all expressions and subexpressions to have an assigned type, not every expression from the earlier untyped lambda calculus can remain valid. The simply typed lambda calculus (Church 1940) ($\lambda^{\to}$) is exactly the restriction of lambda calculus to expressions that are typeable as defined above.

One particularly interesting untypeable expression is the so-called Y combinator, $\lambda f \to (\lambda x \to f\ (x\ x))\ (\lambda x \to f\ (x\ x))$. Since $Y\ f$ evaluates to $f\ (Y\ f)$, it can be used to implement fix-point semantics. In fact, there exist evaluation strategies for $\lambda^{\to}$ that always terminate, implying that $\lambda^{\to}$ is not Turing-complete, and fix-point semantics cannot be implemented using it.

It is possible to extend $\lambda^{\to}$ with a special class of fix-point operators: $fix_\alpha = \lambda f \to f\ (fix_\alpha\ f)$, with type $(\alpha \to \alpha) \to \alpha$. This is enough to make it Turing-complete again.

Even with the above extension, it is not necessary to write the types of expressions in $\lambda^{\to}$, as these can be decidably and efficiently derived using a process called type inference.

3.3.2 System F

Consider the identity function id $= \lambda x \to x$. It can be effectively typed as $Int \to Int$, but it could as well be typed using $(Int \to Int) \to (Int \to Int)$. In general, it is a polymorphic expression, and its type can be anything of the form $\alpha \to \alpha$.

To formalize this, we need type variables. *System F* (Girard 1971) is a less-restricted form of typed lambda calculus. It introduces abstractions and applications over types as well as over values, where type variables are usually written using uppercase characters, and abstractions over them using the uppercase lambda Λ. It is also called polymorphic lambda calculus, or second-order lambda calculus.

The polymorphic identity function is written as $\Lambda A \to \lambda(x :: A) \to x$, and is of type $\forall A.A \to A$. We're (ab)using the Haskell operator "::" here to signify "of type". In more conventional System F notation, it would be written $\Lambda A.\, \lambda x^A.\, x$.

Type inference is not decidable in general for System F, but it is for a restricted version of it, called *Hindley-Milner*.

Also note that typing a Y combinator in System F requires the presence of recursive data types. Without these, or a special fix combinator, System F is not Turing-complete.

3.4 Algebraic data types

One way of encoding more complex data structures, is algebraic data types (ADTs).

ADTs are formally a sum type of product types - that is, a data type whose value can be any of some combinations of element data types.

To use some Haskell code:

$$\begin{aligned} &\textbf{data}\ \mathit{List}\ a = \mathit{EmptyList} \\ &\qquad\qquad\ \ |\ \mathit{Concat}\ a\ (\mathit{List}\ a) \end{aligned}$$

This defines the type $List\ \alpha$ for each α, as either *EmptyList* or as a concatenation (named *Concat*) of an element of type α and another list of type $List\ \alpha$. *EmptyList* and *Concat* are called the constructors here. The fact that the arguments of *Concat* refer back to the complete ADT, makes this an example of a recursive type.

The code above defines *List* as $\Lambda\alpha \to List\ \alpha$, which is in fact a *type constructor*: it maps types to new types. In addition, *EmptyList* has type $\forall\alpha.List\ \alpha$ and *Concat* has type $\forall\alpha.\alpha \to List\ \alpha \to List\ \alpha$.

Algebraic Data Types can be translated to untyped lambda calculus, but it is more often used as a non-function type in typed lambda calculi.

3.5 Type classes

One extension that is allowed by Hindley-Milner, is type classes (Wadler and Blott 1989). These allow one to write values where the implementation depends on the type of the arguments, a property called ad-hoc polymorphism.

Semantically, a type class is comparable to an interface or abstract class in languages like Java or C++. It defines a set of polymorphic values, for which an implementation must be provided by classes that instantiate the type class. Type classes do not define a type by themselves, though.

For example, a type class *Eq* could define a value *eq* of type:

$$\forall a.Eq\ a \Rightarrow a \to a \to Bool$$

For each type that belongs to *Eq*, an implementation for *eq* must be given then. The requirement $Eq\ a$ is called a type constraint. It is given as a predicate *Eq* that applies to types.

Using the polymorphic values defined by a type class, one can define other values that depend on the membership of a type to a given type class, by reusing the type constraint. For example, a function *dist* that counts the number of distinct elements out of two given parameters, can be written as:

$$\lambda x\ y \to ifthenelse\ (eq\ x\ y)\ 1\ 2$$

with type

$$\forall a.Eq\ a \Rightarrow a \to a \to Int$$

Type classes can be converted to System F (after type inference), by replacing type constraints with actual arguments that pass along dictionaries with the implemented functions. In the above example, the dictionary for the membership of type α to type class *Eq*, would have type

$$\alpha \to \alpha \to Bool$$

Using this, the function *dist* above is translated to System F as

$$\Lambda a \to \lambda(d :: a \to a \to Bool)\ (x :: a)\ (y :: a) \to ifthenelse\ (d\ x\ y)\ 1\ 2$$

Where d is the dictionary that is passed along (here a single function). During translation, the type a is known at each specific call site, and is known to belong to the type class *Eq*. This means a dictionary for the instance $Eq\ a$ exists, and is used as implicit argument to the polymorphic function.

3.6 Monads and Functors

To retain referential transparency, the evaluation of an expression should not have any effect beyond its return value. This has multiple advantages:

- Expressions whose return value is never used, do not need to be evaluated (lazy evaluation).
- Evaluation can happen in any order (allowing other strategies than call-by-value).
- Using the same expression a second time can reuse a cached result.
- Correctness proofs and other automated reasoning about programs becomes easier.

3.6.1 Interaction with the real world

Still, interaction with "the real world" is inherently impure. Reading input from a file or from a user will not necessarily result in the same value a second time. Executing a print statement twice is not the same as only executing it once, even though it does not have any meaningful return value.

Several solutions have been used in functional programming languages:

- Fix the evaluation order to call-by-value, and have functions whose evaluation triggers the execution of some impure code. This is done in the ML family of languages (SML (Milner, Tofte, and Macqueen 1997), OCaml (Smith 2006)).
- Use uniqueness typing (Wadler 1991), which extends the type system with values that must be evaluated exactly once, enforced by the compiler. This is for example used in Clean.
- Make the description of side effects explicit, as done using the technique described below.

3.6.2 The $IO\ \alpha$ type

We introduce a type constructor IO. When α is a data type, $IO\ \alpha$ represents a description of an effectful action producing something of type α. The actual representation behind this type is not important, but values of type $IO\ \alpha$ cannot be converted to values of type α. It is a description of a "program" that must be executed or interpreted by a runtime system (causing side effects) to obtain something of type α.

There are three ways of obtaining values of type $IO\ \alpha$:

- Built-in actions that do input/output, for example:
  - *readInt* of type $IO\ Int$. It reads an integer from the user.
  - *writeInt* of type $Int \to IO\ ()$. A function that takes an integer, and returns an action that will write this value to the output. The type $()$ is the unit type, with only a trivial value.
- The *return* function, of type $\forall\alpha.\alpha \to IO\ \alpha$. It creates an action that only produces the argument passed, without performing any side effects.
- The *bind* function, of type $\forall\alpha\ \beta.IO\ \alpha \to (\alpha \to IO\ \beta) \to IO\ \beta$. One way to understand this, is as callbacks in imperative languages. Instead of directly running some user-interfacing code, and using the result, you pass a reference to another piece of code to the user-interfacing code, that will be called when the result is obtained — a paradigm also called continuation passing style (CPS). *bind* does exactly this: it takes an action that produces something of type α, and creates a new action that does the same, but instead of returning its normal result, it is fed to another action first.

Combining different *IO* actions using *bind* allows one to do the same as an imperative program can, given that enough predefined $IO\ \alpha$-types values exist that represent the possible interactions. For example, the following IO action describes reading a number from the user, and outputting it doubled:

$$bind\ readInt\ (\lambda(x :: Int) \to writeInt\ (2 * x))$$

Note that this only describes an interaction with the real world in a purely functional way. Evaluating it does not cause any side effects. One can see execution and evaluation as two separate processes here: the evaluation happening lazily inside the language, to build up a full action of type $IO\,()$, and the execution happening sequentially outside of the language. Of course, execution will typically cause evaluation of the bound functions inside the expression, but this is done in an on-demand and transparent way, invisible from the actual execution.

3.6.3 Monads

In fact, the above interface can be generalized. Whenever a type $M\ \alpha$ exists for each α, such that:

- Represents something that can be considered to "produce" or "contain" one or more values of type α.
- Has meaningful implementations for *bind* and *return*.
- Obeys the laws given below

we call M a *monad* (Wadler 1995). Values of type $M\ \alpha$ are then called monadic actions.

These laws, called monad laws are:

- Left identity: $bind\ (return\ a)\ f \equiv f\ a$
- Right identity: $bind\ m\ return \equiv m$
- Associativity: $bind\ (bind\ m\ f)\ g \equiv bind\ m\ (\lambda x \to bind\ (f\ x)\ g)$

Expressions that combine monadic actions, build up a computation — each action describing a value extended with some decoration.

Supporting monads in general is simply a matter of defining a type class for them, with their required functions *return* and *bind*.

The *Identity* monad The *Identity* monad does not offer any decoration. It simply passes produced values to the next function in a bind. On itself, this is rarely useful, but it is a simple base monad to build upon when using (stacks of) monad transformers.

The *Maybe* monad One simple decoration is the notion of *failure*. This is offered by the *Maybe* type: an action that either produces a value of the designated type, or failure. Failure is propagated through bindings, so that the combined action fails as soon as any of the inner actions in the chain of bindings fail.

The *List* monad There is no requirement that a monadic action is limited to one single produced value. For example, the *List* monad embodies computations that have zero or more results. When binding, each result of the previous action is replaced by the results gotten by feeding it to the bind's second argument.

Monad transformers Multiple monadic decorations can be combined using *Monad transformers* (Liang, Hudak, and Jones 1995). The convention is used to provide a *Name* monad as well as a $Name_T$ monad transformer, where the semantics of the *Name* monad is identical to that of the $Name_T$ transformer applied to the *Identity* monad. That is, *Name* α is a shorthand for $Name_T\ Identity\ \alpha$. Often a $Name_M$ type class is provided as well, implemented by all monad types $t_1\ (Name_T\ (t_2\ \alpha))$.

3.6.4 Functors

A somewhat more general structure that often occurs is that of a *Functor*, representing any structure that can be mapped over — including monads. For the remainder of this text, we consider them together with *applicative* functors (Mcbride

and Paterson 2008), which also require the ability to apply functions inside the structure.

For example, given a function f of type $a \to b \to c \to d$, being applied to arguments $x :: a$, $y :: b$ and $z :: c$, $f\ a\ b\ c$. Often, we want to lift this function to for example be applicable to *Maybe*s instead. Data structures, like *Maybe* for which this can be done mechanically, are called applicative functors.

We require the function *ap* and *return* in a type class:

$$\begin{aligned} ap &:: f\ (a \to b) \to f\ a \to f\ b \\ return &:: a \to f\ a \end{aligned}$$

where *return* coincides with the *return* from monads. For *Maybe*, *ap* is easy to implement: if both arguments contain valid data, the (lifted) application of the first on the second is returned, otherwise, failure.

We define the special syntax $[\![ ]\!]$:

- $[\![f]\!] \equiv return\ f$
- $[\![f\ x]\!] \equiv ap\ (return\ f)\ x$
- $[\![f\ x\ y]\!] \equiv ap\ (ap\ (return\ f)\ x)\ y$
- $[\![f\ x\ y\ z]\!] \equiv ap\ (ap\ (ap\ (return\ f)\ x)\ y)\ z$
- ...

Thus, $[\![f\ x\ y\ z]\!]$ can be considered the lifted version of $f\ x\ y\ z$: it now takes lifted arguments (*Maybe* a, *Maybe* b and *Maybe* c instead of a, b and c), and returns a lifted result (*Maybe* d instead of d).

3.7 Haskell

In what follows, the functional language *Haskell* (Peyton Jones et al. 2003) will be used. It combines the features listed above. It is a lazy, statically strongly typed language, whose semantics is based on the lambda calculus.

3.7.1 Basics

Algebraic Data Types It offers algebraic data types, and a special syntax to decompose them: pattern-patching. Some ADTs are predefined in the language:

```
data Bool = False
          | True

data Maybe a = Nothing
             | Just a
```

One special ADT is predefined for (linked) lists. In pseudo-syntax:

$$\begin{aligned}&\textbf{data}\ [\,]\ a = [\,]\\ &\qquad\qquad \mid\ a : ([\,]\ a)\end{aligned}$$

The types $[\,]\ a$ and $[a]$ are identical, and refer to the type of lists of a. The value $[\,]$ represents the empty list, and (:) is a constructor that combines an element of type a with another list of type $[\,]\ a$. Syntactic sugar is provided that e.g., allows $a : (b : [\,])$ to be written as $[a, b]$.

The following Haskell code defines the ADT *Nat* for Peano natural numbers, and an *isZero* function for it:

$$\begin{aligned}&\textbf{data}\ \mathit{Nat} = \mathit{Zero}\\ &\qquad\qquad \mid\ \mathit{Succ}\ \mathit{Nat}\\ &\mathit{isZero} = \lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\\ &\quad \mathit{Zero}\ \ \rightarrow \mathit{True}\\ &\quad \mathit{Succ}\ y\ \rightarrow \mathit{False}\end{aligned}$$

Pattern matching can be used directly in the definition of functions, improving notation:

$$\begin{aligned}&\mathit{isZero}\ \mathit{Zero}\ \ \ \ \ \ = \mathit{True}\\ &\mathit{isZero}\ (\mathit{Succ}\ y) = \mathit{False}\end{aligned}$$

Implicit fix-point The Haskell type system is Hindley-Milner, which allows type inference. Some extensions, including recursive type definitions and an implicit fix-point operator are available. The following code defines an addition function for the above Peano numbers:

$$\begin{aligned}&\mathit{peanoAdd}\ \mathit{Zero}\ y\ \ \ \ \ \ = y\\ &\mathit{peanoAdd}\ (\mathit{Succ}\ x)\ y = \mathit{Succ}\ (\mathit{peanoAdd}\ x\ y)\end{aligned}$$

As this function refers to its own definition, it is to be understood as the application of an implicit fix-point operator:

$$\begin{aligned}&\mathit{peanoAdd}' = \lambda \mathit{self}\ x\ y \rightarrow \textbf{case}\ x\ \textbf{of}\\ &\quad \mathit{Zero}\ \ \ \ \ \rightarrow y\\ &\quad (\mathit{Succ}\ x) \rightarrow \mathit{Succ}\ (\mathit{self}\ x\ y)\\ &\mathit{peanoAdd} = \mathit{fix}\ \mathit{peanoAdd}'\end{aligned}$$

Explicit type signatures for definitions can be provided, using "::":

$$\begin{aligned}&\mathit{isZero}\ \ \ \ \ \ :: \mathit{Nat} \rightarrow \mathit{Bool}\\ &\mathit{peanoAdd}\ \ :: \mathit{Nat} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat}\\ &\mathit{peanoAdd}' :: (\mathit{Nat} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat}) \rightarrow \mathit{Nat} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat}\end{aligned}$$

Monads Haskell supports polymorphic definitions and type classes, and instances for types can be made arbitrarily. Interaction with the real world is done through an *IO* type as described above. A type class for monads is predefined with functions *return* and the $\gg=$ operator for *bind*:

$$\begin{array}{l}\textbf{class } \mathit{Monad}\ m\ \textbf{where}\\ \quad \mathit{return} :: a \to m\ a\\ \quad (\gg=)\ \ :: m\ a \to (a \to m\ b) \to m\ b\end{array}$$

It includes the types *Identity*, *IO*, *Maybe*, and the list type $[\alpha]$. Special syntactic sugar is provided to write monadic actions, called the *do notation*:

$$\begin{array}{l}\textbf{do } x \leftarrow \mathit{readInt}\\ \quad\ \ \mathit{writeInt}\ (2 * x)\end{array}$$

This would be desugared into:

$$\begin{array}{l}\mathit{readInt} \gg= \quad (\lambda x \to\\ \mathit{writeInt}\ (2 * x))\end{array}$$

Given that *readInt* and *writeInt* are of type $\mathit{IO}\ \mathit{Int}$ and $\mathit{Int} \to \mathit{IO}\ ()$ respectively, this defines a combined action of type $\mathit{IO}\ ()$ that reads a number and outputs its double.

Functors Applicative functors are available, split over the type classes *Functor* and *Applicative*. The operator <*> corresponds to *ap*, <$> does the same, but takes a pure function instead of a lifted one as argument, and *return* for functors is called *pure*:

$$\begin{array}{lll}(<*>) :: \mathit{Applicative}\ f \Rightarrow & f\ (a \to b) \to f\ a \to f\ b\\ (<\$>) :: \mathit{Functor}\ f \quad \Rightarrow & (a \to b) \to f\ a \to f\ b\\ \mathit{pure} \quad :: \mathit{Functor}\ f \quad \Rightarrow & a \to f\ a\end{array}$$

Using these, one can easily write a lifted expression:

- $[\![f]\!]$ is written as $\mathit{pure}\ f$
- $[\![f\ x]\!]$ is written as $f\ <\$>\ x$
- $[\![f\ x\ y]\!]$ is written as $f\ <\$>\ x\ <*>\ y$
- $[\![f\ x\ y\ z]\!]$ is written as $f\ <\$>\ x\ <*>\ y\ <*>\ z$
- ...

Is everything a function? One common misconception about Haskell, being a functional language, is that "everything is a function"[2], and values of non-function types can be considered "functions with zero arguments". Although it is possible to define things this way, it is an oversimplification and possibly confusing. For example, what arity is the "function" $id :: \forall a.a \to a$? One? What if a is instantiated to $Int \to Int$, turning the entire expression into something of type $(Int \to Int) \to (Int \to Int)$, which is identical to $(Int \to Int) \to Int \to Int$, a function of two arguments?

It is closer to reality to define functions as values of a function type $(a \to b)$, always having one argument and one return value — though the return value may be a function again. This way, not every value is a function, but — consistent with the paradigm of first-class functions — every function is a value.

3.7.2 Predefined operators and functions

Due to the rich type system, and polymorphism, some very general operators are possible and predefined in Haskell's standard library. As they will be used in the code examples throughout the rest of this text, here is a summary:

$$(\$) :: (a \to b) \to a \to b$$
$$(\$) = id$$

The function application operator \$ is identical to the normal function application (designated by simple whitespace). It has, however, different order of precedence when parsing. This allows expressions like

$$f\ x\ (g\ y\ (h\ z))$$

to be written as

$$f\ x\ \$\ g\ y\ \$\ h\ z$$

The function composition operator $\circ$ combines functions without needing an explicit argument.

$$(\circ) :: (b \to a) \to (c \to b) \to (c \to a)$$
$$(f \circ g)\ x = f\ (g\ x)$$

This allows the above expression to also be written as

$$(f\ x \circ g\ y \circ h)\ z$$

or as

[2] See http://conal.net/blog/posts/everything-is-a-function-in-haskell

$$f\ x \circ g\ y \circ h\ \$\ z$$

$$const :: a \to b \to a$$
$$const\ x\ \_ = x$$

The *constant* function *const* turns a value into a function that always returns that value. For example,

$$f\ x = 5$$

can be written as

$$f = const\ 5$$

$$(\gg) :: Monad\ m \Rightarrow m\ a \to m\ b \to m\ b$$
$$x \gg y = x \gg\!= (\backslash\_ \to y)$$

The shortcut binding operator $\gg$ is identical to the normal binding operator $\gg\!=$, except that it ignores the produced value of the first argument.

3.7.3 Strength for DSLs

As stated in Section 1.2, Haskell is an appropriate host language for EDSLs. We recapitulate the relevant features.

User-defined operators Libraries in Haskell can define custom operators — even ones not defined by the language, as long as they consist of a set of predefined characters. These behave exactly like any other (top-level) definition, except they are written in between the arguments (infix) instead of in front of them (prefix). They can be defined within type classes, and used as curried functions as well.

Strong type system The strong type system, including an extension called Generalized Algebraic Data Types (Cheney and Hinze 2003), allow other languages' constructs and their types to be embedded in Haskell's type system. This means the EDSL language elements can be typed, while using the Haskell compiler's features to provide type inference and type checking.

Lazy execution Because Haskell is a lazy language, EDSL "blocks" in the code can evaluate to an intermediate data structure, with actual execution happening on-demand.

Part II

Constraint Modeling

Chapter 4

Introduction

Constraint Modeling is the process of writing a constraint model for CP. This includes defining variables, initial domains, and imposing constraints on them. In the context of this work, we consider it separate from the modeling of the search tree and its exploration strategy, which is covered in Part III.

This CP approach abstracts from the operational details of finding a solution. However, often there are still many different ways in which a given problem can be modeled as a CP problem. These different representations may have different performance for solving. A more declarative approach is often wanted that abstracts from many uninteresting details. To overcome this, several systems exist that incorporate CP. As an additional advantage, these bring some abstractions and chances for code reuse to the modeling process. Among these systems are languages specifically designed for CP and related paradigms, such as Zinc (Marriott et al. 2008), Comet (Van Hentenryck and Michel 2005), OPL (Van Hentenryck 1999) and Essence (Frisch, Grum, Jefferson, Hernndez, and Miguel 2005).

As explained in Section 1.1, they often run behind on state of the art in programming languages. The alternative, libraries or APIs for solving CP problems in existing languages are often far less declarative and require boilerplate. One state-of-the-art solver with bindings in many languages is Gecode (Gecode Team 2006).

We try to combine the advantages of both through the use of an EDSL in Haskell. This chapter explains the design of the Monadic Constraint Programming framework (MCP) (Schrijvers, Stuckey, and Wadler 2009), which we use to build our EDSL and supporting system. MCP itself is very generic, and supports more search-based back-ends than just CP. Our EDSL itself is however specific for CP solving of finite domain (FD) problems, and will be described in the next three chapters.

By integrating CP in an existing language as an EDSL, we directly obtain state-of-the-art functional programming support with zero effort. This has a con-

Code listing 4.1 Base *Solver* class

```
class Monad solver ⇒ Solver solver where
  type Constraint solver :: ∗
  add :: Constraint solver → solver ()
  isFailed :: solver Bool
  run :: solver a → a
```

siderable advantage compared to special-purpose Functional Constraint (Logic) Programming (FCP) languages such as Alice (Kornstaedt 2001), Curry (Hanus, Kuchen, and Moreno-Navarro 1995) or TOY (Fernandez, Hortala-Gonzalez, Saenz-Perez, and Del Vado-Virseda 2007).

4.1 Design of MCP

MCP's design consists of a back-end that is independent from the actual solvers, constraints or type of variables used. Solver-specific front-ends extend the functionality with actual variables and constraints.

4.1.1 The *Solver* class

As described in Section 2.3, the propagation part of constraint solving corresponds to a rewriting of the constraint problem $\langle X, D, C\rangle$ to an equivalent problem $\langle X, D', C\rangle$ with smaller domains. In practice, optimizations also remove implied constraints or superfluous variables from the problem, turning it into a process that transforms $\langle X, D, C\rangle$ into $\langle X', D', C'\rangle$.

For a constraint solving process, the actual constraints do not need to be observable, only whether they bring the solver in an unsatisfiable state ("failure"). Therefore, we define a solver as an instance of the *Solver* type class, shown in Code listing 4.1.

First of all, this definition requires a type *solver* to at least be a monad. It is typically implemented by the *State* monad or $State_T$ monad transformer, but its state does not need to be inspectable itself either, and can thus be modeled as a black box.

The second line defines an *associated type*: *Constraint* here represents a type function, mapping each *solver* to the associated type *Constraint solver*. Each instance must define that type as part of its declaration. The next two lines define members of the type class. *add* adds a constraint to the solver's internal state. *isFailed* checks whether the solver is in a failed state. *run* is used to run the actual

solver computation, and extract a result.

When doing search, and failure occurs, it is necessary to backtrack to a previous state. To avoid forcing *Solver* implementations to expose the type of their internal state, we introduce another associated type, and corresponding member definitions:

```
class Monad solver ⇒ Solver solver where
  ...
  type Label solver :: ∗
  mark :: solver (Label solver)
  goto :: Label solver → solver ()
```

Label solver is the type used by *solver* to represent its checkpoints. *mark* creates a checkpoint for its current state, and *goto* makes the solver return to such a previous state.

4.1.2 The *Term* class

With the above *Solver* definition, there is no interface for creating variables to enforce constraints over. As multiple variable types may be meaningful for a single solver, we put them in a separate type class: *Term*. *Term s t* expresses that *t* is a term type of solver type *s*. Each term type provides a method *newvar* to generate new constraint variables of that type:

```
class Solver solver ⇒ Term solver term where
  newvar :: solver term
```

Specific implementations may additionally provide interfaces to inspect variables' domains or other properties. A general interface for doing so will be given in Section 4.1.6.

4.1.3 A trivial search

Code listing 4.2 shows a first full search algorithm. It uses a slightly richer interface. The constraint problem is assumed to be given as an initial *Solver* action *init*, and an action *extract* :: *Solver s* ⇒ *s* (*Maybe r*) is expected, which:

- Checks whether the solver's state represents a solution, and if so returns *Just* that solution
- Otherwise return *Nothing*

Additionally, we need a function to do branching: given an unsolved but not failed solver state, create a list of mutually-exclusive constraints that restrict variables' domains: *branch* :: *Solver s* ⇒ *s* [*Constraint s*].

Code listing 4.2 Basic depth-first search routine

```
dfs :: Solver s ⇒ s () → s (Maybe r) → s [Constraint s] → [r]
dfs init extract branch = run (init ≫ solve)
  where solve = do
                  failed ← isFailed
                  if failed
                    then return []
                    else solveGood
        solveGood = do sol ← extract
                       case sol of
                         Just res → return [res]
                         Nothing → solveBranch
        solveBranch = do trunk ← mark
                         branches ← branch
                         sols ← mapM (solveFrom trunk) branches
                         return $ concat sols
        solveFrom l c = goto l ≫ add c ≫ solve
```

To find solutions, a combined *Solver* action, (*init* ≫ *solve*) is created that initializes, and then searches for solutions using the resulting state. This combined action is then passed to *run* to retrieve the actual solutions as a list.

Search is split up in three cases. First, a check for failure is done. If this is the case, there are no solutions, and [] is returned. Otherwise, execution continues in *solveGood*. It checks whether a solution has been reached, and if so, returns it. Otherwise, control is directed to *solveBranch*, which creates a label for the current state (*trunk*), and a list of mutually-exclusive constraints (*branches*) which correspond to the different branches to be created. Using the generic monadic function *mapM*, we restart a search procedure for each branch at the trunk, by adding the respective constraint and calling *solve* recursively.

Depending on the *solver* implementation, the resulting solution list may be built up lazily, producing additional solutions as demanded by whatever code is consuming the values in the list. This is a very powerful feature, as not all solutions are always necessary, and thus do not necessarily need to be searched for. Because of laziness, this on-demand search is completely hidden from the code using it.

Code listing 4.3 The *SearchTree s a* type

```
data SearchTree s a =
    Fail
  | Solution a
  | Branch [SearchTree s a]
  | Add (Constraint s) (SearchTree s a)
  | Dynamic (s (SearchTree s a))
```

4.1.4 The *SearchTree* type

The above *dfs* function is simple and straightforward. However, it lacks flexibility, as it is only able to do depth-first search, and there is no way to interact with the solver state once the search process takes off.

Ideally, we would like to interact with the search tree itself. We introduce the *SearchTree* type to represent it. The tree type is parametrized in the constraint solver s and solution type a. This provides a type-safe way for representing constraint problem models for arbitrary solvers and solution types.

We use an algebraic data type. Its values represent nodes of a generalized tree structure, with leaf nodes that correspond to failure (*Fail*) or a found solution (*Solution*). Internal nodes correspond to either a branching, or a constraint to be active in a subtree.

We need a construct to mimic the *branch* argument to *dfs* however: an action that has access to the solver state, and depending on it, create branches. To this end there is a *Dynamic* constructor in *SearchTree*. It wraps a solver action that itself produces a subtree. During evaluation, when the *Dynamic* node is being explored, its action is bound, and evaluation continues with the produced subtree. This subtree could then contain the necessary *Branch* nodes, each containing an *Add* node.

This allows us to write a constraint problem as a single initial *SearchTree* (ignoring the introduction of new variables for now), and retrieve its solutions. Both the *extract* and *branch* arguments to *dfs* can now be specified as *Dynamic* nodes in the tree.

One can see *SearchTree s* also fits the form of a *Monad*, as shown in Code listing 4.4, with the *Solution* nodes representing the produced values, and binding meaning replacing solutions with subtrees. A failed tree bound to anything remains failed. Providing an instance for *Monad* (*SearchTree s*) facilitates writing actions of type SearchTree. As *SearchTree*s will be the data type used to write constraint problems, we will call its values *model trees*.

The *liftTree* function "lifts" a computation on the *Solver* level, to a *SearchTree*

Code listing 4.4 Monad instance for *SearchTree s* and auxiliary operations

```
instance Solver s ⇒ Monad (SearchTree s) where
  return a          = Solution a
  Fail         >>= f = Fail
  Solution a   >>= f = f a
  Branch lst   >>= f = Branch (map (>>=f) lst)
  Add c s      >>= f = Add c (s >>= f)
  Dynamic x    >>= f = Dynamic (x >>= return ∘ (>>=f))

liftTree :: Solver s ⇒ s a → SearchTree s a
liftTree s = Dynamic (s >>= return ∘ return)

true :: Solver s ⇒ SearchTree s ()
true = Solution ()

addC :: Solver s ⇒ Constraint s → SearchTree s ()
addC c = Add c true
```

that produces the same value. It requires two *return* calls: the first one to go from *SearchTree s a* to *s* (*SearchTree s a*), the second one to go from *a* to *SearchTree s a*.

The *true* function is a shorthand for producing the () unit. It is used as an identity in chains of bindings.

The search tree corresponding to the arguments *extract* and *branch* to *dfs* is shown in Code listing 4.5. *baseTree* first checks whether the solver is in a failed state, in which case a *Fail* node is returned. In the normal case, the *true* identity is employed. The following statements within the do-construct do not affect the *Fail*. Afterwards, *extract* is called to check whether a solution was found. If so, it is returned, and otherwise, constraints for the branches are requested, and each is turned into a full *SearchTree s r* by *add*ing it, followed by a recursive call to *baseTree*.

Given that all search tree logic is now bundled in *SearchTree*, very generic traversal routines can be implemented. Code listing 4.6 shows a minimal depth-first *SearchTree* traverser.

In fact, the *init* argument is superfluous - it can be incorporated in *SearchTree* as well. Building the initial solver state consists of two parts: the introduction of new variables, and adding constraints over them. The only missing part is adding new variables. Simple ADTs are not powerful enough to express this, so we turn to *Generalized* ADTs, or GADTs (Cheney and Hinze 2003), as shown in Code listing 4.7. These allow explicit type declarations for each constructor.

A new constructor, *NewVar*, was added here: it takes a function that maps a

Code listing 4.5 Basic search tree generator

```
baseTree :: Solver s ⇒ s (Maybe r) → s [Constraint s] → SearchTree s r
baseTree extract branch =
  do failed ← lift isFailed
     if failed
       then Fail
       else true
     sol ← lift extract
     case sol of
       Just res → Solution res
       Nothing → do branches ← lift branch
                    Branch (map subtree branches)
where subtree c = do addC c
                     baseTree extract branch
```

Code listing 4.6 Depth-first *SearchTree* traverser

```
dfsSearch :: Solver s ⇒ s () → SearchTree s a → [a]
dfsSearch init tree = run (init ≫ search tree)
  where search Fail          = return []
        search (Solution x)  = return [x]
        search (Branch lst)  = do trunk ← mark
                                  sols ← mapM (searchFrom trunk) lst
                                  return $ concat sols
        search (Add c sub)   = add c ≫ search sub
        search (Dynamic x)   = x ⋙= search
        searchFrom label x   = goto label ≫ search x
```

Code listing 4.7 Full definition of *SearchTree s a*

```
data SearchTree s a where
  Fail      :: SearchTree s a
  Solution  :: a → SearchTree s a
  Branch    :: [SearchTree s a] → SearchTree s a
  Add       :: Constraint s → SearchTree s a → SearchTree s a
  Dynamic   :: s (SearchTree s a) → SearchTree s a
  NewVar    :: Term s t ⇒ (t → SearchTree s a) → SearchTree s a
```

Code listing 4.8 Syntactic sugar for *SearchTree* manipulation

```
exists :: (Solver s, Term s t) ⇒ SearchTree s t
exists = NewVar Solution

exist :: (Solver s, Term s t) ⇒ Int → SearchTree s [t]
exist n = mapM (const exists) [1 .. n]

(∧) :: Solver s ⇒ SearchTree s () → SearchTree s a → SearchTree s a
(∧) = (≫)

(∨) :: Solver s ⇒ SearchTree s a → SearchTree s a → SearchTree s a
a ∨ b = Branch [a, b]

conj :: Solver s ⇒ [SearchTree s ()] → SearchTree s ()
conj list = foldr (∧) (Solution ()) list

disj :: Solver s ⇒ [SearchTree s a] → SearchTree s a
disj list = foldr1 (∨) list
```

newly instantiated variable to the search tree over it. The GADT notation was necessary to be able to write the type constraint *Term s t*. This encoding prevents using a variable outside of the part of the tree where it is defined.

On the other hand, it still allows a general implementation for code that iterates the search tree, by calling the term type's *newvar* function each time a *NewVar* node is traversed.

4.1.5 Syntactic sugar

On top of the model data type, MCP provides syntactic sugar (functions that construct model trees). Code listing 4.8 shows the following definitions:

- *exists*: A *SearchTree* that produces a new (constraint) variable

- *exist n*: Introduce n new variables
- $(\wedge)$: Create a conjunction of two *SearchTree*s
- $(\vee)$: Create a disjunction of two *SearchTree*s
- *conj*: Create a conjunction of a list of *SearchTree*s
- *disj*: Create a disjunction of a list of *SearchTree*s

The helper function *exists* is useful inside monadic do-blocks for the definition of SearchTree actions, as for example:

```
do v ← exists
   addC (someConstraint v)
   baseTree extract branch
```

is, after application of the $(\gg=)$ definition for *SearchTree*s, identical to:

```
NewVar (λv → do addC (someConstraint v)
                baseTree extract branch
       )
```

It is called *exists* because it introduces a new variable that corresponds to an existentially quantified variable in the mathematical description of the problem.

4.1.6 Labeling

The above *Term* class is rich enough to introduce new variables in models, but it provides no way to inspect variables and enumerate solutions.

An additional type class *EnumTerm* is introduced to expose this functionality, shown in Code listing 4.9. We keep it separate from *Term*, as it is of a higher level:

- *TermBase s t* represents the type of values the constraint variable represents. For an variable type t representing integers, *TermBase s t* is probably *Int*.
- *getDomainSize* returns a measure for the remaining size of the domain. This can be the actual size if that is tracked by the solver, or some number whose values changes monotonically with the actual size. In case of failure, 0 is expected, and in the case a variable is assigned, 1.
- *getDomain* returns the actual values left in the domain of variable.
- *setValue* produces a set of constraints that together assign a given value to a variable.

Code listing 4.9 *EnumTerm* definition

```
class (Solver s, Term s t) ⇒ EnumTerm s t where
  type TermBase s t :: ∗
  setValue :: t → TermBase s t → s [Constraint s]
  getDomain :: t → s [TermBase s t]
  getDomainSize :: t → s Int
  getValue :: t → s (Maybe (TermBase st))
  defSplit :: t → s [[Constraint s]]
  defOrder :: [t] → s [t]
```

- *getValue* requests a variable's value, if it is already assigned.
- *defSplit* provides a default branching, which splits the domain of the variable. It creates a list of sets of constraints, each of which together forms one branch.
- *defOrder* provides a default variable ordering.

Only *setValue* and *getDomain* are required: for the others, default implementations can be given. The reason for placing them within the type class is to allow solvers to override the defaults.

Given a list of variables, we can now write *extract* and *branch* functions for *baseTree*:

```
extract :: EnumTerm s t ⇒ [t] → s (Maybe [TermBase s t])
branch :: EnumTerm s t ⇒ [t] → s [Constraint s]
```

Combining these with *baseTree*, we get a default labeling routine:

```
label :: EnumTerm s t ⇒ [t] → s [TermBase s t]
label vars = do
  orderedVars ← defOrder vars
  baseTree (extract orderedVars) (branch orderedVars)
```

Which, given a list of variables, produces a search tree that returns their assignment.

Example Without going into details of the actual solver, this short example shows how to use the constructs shown above. Assume we want to find two numbers whose product is 15, and a constraint *CMult* v_1 v_2 n exists, which constrains the product of variables v_1 and v_2 of type *MyTerm* to n in solver *MySolver*.

$$
\begin{aligned}
&\mathit{model} :: \mathit{SearchTree}\ \mathit{MySolver}\ [\mathit{MyTerm}]\\
&\mathit{model} = \mathbf{do}\\
&\quad a \leftarrow \mathit{exists}\\
&\quad b \leftarrow \mathit{exists}\\
&\quad \mathit{addC}\ \$\ \mathit{CMult}\ a\ b\ 15\\
&\quad \mathit{label}\ [a, b]\\
&\mathit{main} = \mathit{print}\ \$\ \mathit{dfsSearch}\ \mathit{model}
\end{aligned}
$$

The actual constructed *SearchTree* would be

$$\mathit{NewVar}\ (\lambda v_1 \rightarrow \mathit{NewVar}\ (\lambda v_2 \rightarrow \mathit{Add}\ (\mathit{CMult}\ v_1\ v_2\ 15)\ (\mathit{label}\ [v_1, v_2])))$$

label here corresponds to a *Dynamic* node, which contains code that generates the rest of the search tree based on the run-time solver state. During the first iteration in *dfsSearch*, this $\mathit{label}\ [v_1, v_2]$ produces the subtree

$$
\begin{aligned}
&\mathit{Try}\ (\mathit{Add}\ (\mathit{CLabel}\ v_1\ \ldots)\ (\mathit{label}\ [v_1, v_2]))\\
&\quad\ (\mathit{Add}\ (\mathit{CLabel}\ v_1\ \ldots)\ (\mathit{label}\ [v_1, v_2]))
\end{aligned}
$$

where $\mathit{CLabel}\ v_1\ \ldots$ is some hypothetical constraint that shrinks the domain of v_1. The left branch will be explored first, producing similar subtrees (though with different *CLabel* constraints). Eventually, either a conflict is reached, in which case the next call to label produces

$$\mathit{Fail}$$

and *dfsSearch* returns to an earlier branch, and continues there. In the alternative case, where an assignment to v_1 is found, *label* produces instead:

$$
\begin{aligned}
&\mathit{Try}\ (\mathit{Add}\ (\mathit{CLabel}\ v_2\ \ldots)\ (\mathit{label}\ [v_2]))\\
&\quad\ (\mathit{Add}\ (\mathit{CLabel}\ v_2\ \ldots)\ (\mathit{label}\ [v_2]))
\end{aligned}
$$

which also branches until an assigned is reached for v_2, meaning that a solution has been found. At that point, *baseTree* (as called from *label*) receives a successful result from *extract*, and produces the node:

$$\mathit{Solution}\ [5, 3]$$

which is collected by *dfsSearch* and finally returned as one of the solutions.

4.2 Implementation of MCP

The previous section explained the design of MCP by building an alternative implementation step by step. The actual implementation of the library as it existed before the work presented in this thesis began, differs somewhat and is more extensive. In this section we highlight the differences with the system as it was presented in (Schrijvers, Stuckey, and Wadler 2009).

Search transformers Instead of the *dfsSearch* function explained above, MCP comes with a more general *Search Transformer* framework. As it is largely superseded by Search Combinators (Chapter 9) — which will be explained more thoroughly in part III —, it is only discussed briefly here.

The basic idea is that search is performed by maintaining a set (queue) of search tree nodes that need exploration, and then processing them one by one. The node selection is controlled by choosing a queue type (stack, FIFO, random, ...). The actual processing is controlled through a combination of transformers that modify the behavior of several hooks. Examples of transformers include:

- Branch and bound
- Restarting search
- Limited discrepancy search (Harvey and Ginsberg 1995)
- Node-bounded search

Miscellaneous

- The *Term* interface is separate from the *Solver* interface. Using another type class allows for several term types supported by a single solver.
- Several operations for dealing with labeling are grouped together in *EnumTerm*. This allows solvers to have specific default labeling orderings.
- The type *Tree* has been renamed to *SearchTree*.
- *exists* is of type *SearchTree s t*, instead of $(t \to \mathit{SearchTree}\ s\ a) \to \mathit{SearchTree}\ s\ a$, allowing variables to be instantiated using $v \leftarrow \mathit{exists}$ instead of using $\mathit{exists}\ \$\ \lambda v \to \ldots$.

Basic FD solver In the next chapters we will develop an extensive general framework for modeling FD problems in MCP. The original MCP library did ship with a simple FD solver and interface already, however. Using that interface, one needed to add the relevant constraints manually to the system. For example, the initial model for the grocery puzzle — where four numbers a, b, c and d, represented as multiples of 0.01, have both their sum and product equal to 7.11 — would be written as:

```
grocery = exist 4 $ λ[a, b, c, d] → do
  addC (FDEq (a .+. b .+. c .+. d) 711)
  addC (FDEq (a .*. b .*. c .*. d) 711000000)
  ...
```

where *FDEq expr val* is one of the constraints of the provided solver. Clearly, a more concise and declarative interface would be preferable.

Chapter 5

FD-MCP

5.1 Introduction

The FD-MCP framework introduces an extra layer of abstraction between the more generic *Solver* interface of the MCP framework and the concrete solver implementations.

In contrast to MCP's generic *Solver* interface, which is parametric in the constraint domain, the interface of FD-MCP is fully aware of the finite domain (FD) constraint domain: both its syntax (terms and constraints) and meaning (constraint theory). It does however make abstraction of the particular FD solver and propagation techniques used. It provides a uniform modeling language that hides the syntactic differences between different FD solvers.

On the one hand, this allows the development of solver-independent models, model transformations (e.g., for optimization) and model abstractions (capturing frequently used patterns). On the other hand, specific solvers may focus on the efficient processing of their constraint primitives without worrying about modeling infrastructure.

A front-end layer integrates with MCP's *SearchTree* for writing models. An intermediate layer performs common optimizations and transformations. Several back-ends can plug into the rest of the framework to provide mappings to actual underlying solvers. We will refer to problems written using FD-MCP's uniform language as "high-level models".

5.2 General architecture

In order to plug into MCP's *SearchTree*, but still provide a single syntax, we wrap the underlying *Solver* monad.

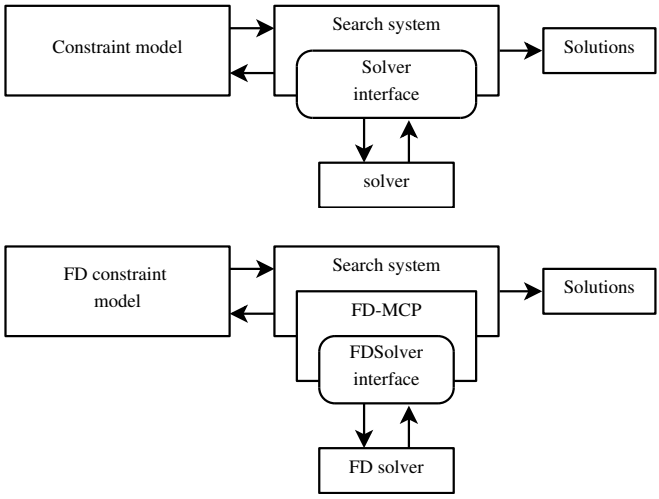

Figure 5.1: General MCP architecture, with and without FD layer

newtype *FDWrapper s a* = *FDWrapper* (*s a*)

This introduces the *FDWrapper* type. The **newtype** keyword works similar to **data**, but it supports only a single constructor with a single argument. Internally however, values of the newly created type are stored identically to values of type *s a*. This means that the wrapping only exists at the type level, and not on the value level. It does provide less laziness though, in terms of Haskell semantics, as evaluation of the wrapped value implies immediate evaluation of the inner value.

Since it is a separate type, we can provide our own instances for *Solver*, as shown in Code listing 5.1 It requires that *s* is already a *Solver*. It uses the type *FDConstraint* as constraint, whose internals will be explained later. However, it can simply reuse the labels provided by the actual solver underneath, and the related functions are simply wrapped as well. The implementation of *add* is problematic though: as will be shown in Section 5.4.4, the translation process requires a state shared for the entire high-level model. We therefore turn to a more complex wrapping:

newtype *FDWrapper s a* = *FDWrapper* (*State*$_T$ (*FDState s*) *s a*)
deriving (*Monad*, *State*$_M$ (*FDState s*))

FDWrapper s is now a wrapper around a solver, transformed by *State*$_T$. This allows threading an internal state of type *FDState s* along with the solver state. The actual definition of *FDState s* will be given later.

Code listing 5.1 Basic *Solver* instance for *FDWrapper s*

```
instance Solver s ⇒ Solver (FDWrapper s) where
  type Constraint (FDWrapper s) = FDConstraint
  type Label (FDWrapper s)      = Label s
  add c    = ...
  isFailed = wrap isFailed
  run      = wrap run
  mark     = wrap mark
  goto l   = wrap (goto l)
wrap :: Solver s ⇒ s a → FDWrapper s a
wrap = FDWrapper
```

For convenience, we support both the constraints of the underlying solver directly, as well as high-level general finite domain constraints of type *FDConstraint*. To that end, we use the *Either a b* type:

```
data Either a b = Left a
                | Right b
```

Thus, for *Constraint* (*FDWrapper s*) we use *Either FDConstraint* (*Constraint s*).

As *FDWrapper*-wrapped solvers have two layers of state, both need to be stored in its labels. The *FDLabel s* type is used to store them:

```
data FDLabel s = FDLabel (FDState s) (Label s)
```

The actual *Solver* instance used is shown in Code listing 5.2. The code for interacting with the real solver, including calls *run*, *add* and *isFailed*, will be given later. Adding a constraint for the underlying solver is easy though. To create a label, an *FDWrapper*-wrapped action is used. It retrieves a label for the underlying solver using *lift mark* (*lift* lifts an expression from type *s a* to $State_T$ (*FDState s*) *s a*, here). Additionally, the state of the $State_T$ layer is retrieved using *get*, and both are returned in a combined *FDLabel*. For backtracking to a previous state, we do something similar: reset the outer state using *put*, and reset the inner state using a lifted *goto*.

Given that each *FDWrapper s* is a *Solver*, *SearchTree*s can be built on top of it. As these are very common, we introduce a shorthand for them:

```
type FDModel s a = SearchTree (FDWrapper s) a
```

Table 5.1 gives a summary of the solver action types.

Code listing 5.2 *Solver* instance for *FDWrapper s*

```
instance Solver s ⇒ Solver (FDWrapper s) where
  type Constraint (FDWrapper s) = Either FDConstraint (Constraint s)
  type Label (FDWrapper s)      = FDLabel s
  run                 = ...
  add (Left m)        = ...
  add (Right c)       = wrap (add c)
  isFailed            = wrap isFailed
  mark                = FDWrapper $ do
                          iLabel ← lift mark
                          oLabel ← get
                          return $ FDLabel oLabel iLabel
  goto (FDLabel o i) = FDWrapper $ do
                          put o
                          lift $ goto i
wrap :: Solver s ⇒ s a → FDWrapper s a
wrap = FDWrapper ∘ lift
```

	solver action	model tree
basic solvers	*s a*	*SearchTree s a*
FD-wrapped solvers	*FDWrapper s a*	*FDModel s a*

Table 5.1: Solver action types

Code listing 5.3 Grocery example in FD-MCP

```
grocery = do
  list@[a, b, c, d] ← exist 4
  list `allin` (0, 711)
  a + b + c + d @≡ 711
  a * b * c * d @≡ 711000000
  a @⩾ b
  b @⩾ c
  c @⩾ d
  return list
```

5.3 Front-end syntax

The paradigm that FD-MCP follows, is that the collection of constraints in a problem is represented by predicates, i.e., Boolean expressions that evaluate to true for solutions. This expression refers to the constraint variables, which are as much as possible treated like normal (known) values.

Code listing 5.3 shows an implementation of the grocery example of Section 4.2 for FD-MCP. Four integer variables are instantiated using *exist*, and bound to the variable named *list*, with the individual elements aliased as a, b, c and d. The *allin* constraint is used here in infix form (‘*allin*‘) to delimit their domains. The next two lines give the actual sum and product constraints, and the final three lines force the values to be of increasing value — breaking the inherent symmetry in the problem representation (Gent and Smith 2000). The last line returns the list.

Code listing 5.4 shows a more advanced example, Sudoku. It represents the array of $9 * 9$ field variables using a Haskell list, created using *exist*. ‘*allin*‘ is used to limit the domain of all list elements to $[1 \,..\, 9]$. The next three lines define local functions that extract list variables corresponding to the various rows, columns and $(3 * 3)$ blocks in the sudoku field. The following 5 lines use the standard *forM* function to repeat the monadic action given by the do-block multiple times. The *allDiff* function invokes the ALLDIFFERENT constraint. A do block which posts several constraints, is considered equivalent to the conjunction of these constraints.

5.3.1 Model expressions

Internally, Boolean expressions are used to represent the constraint in a high-level fashion. These are represented as an expression tree — corresponding to the abstract syntax tree of the high-level model, defined by the GADT given in Code

Code listing 5.4 Sudoku example in FD-MCP

```
sudoku =
  do mat ← exist 81
     mat `allin` (1, 9)
     let row i = [mat !! (i * 9 + p) | p ← [0 .. 8]]
         col i = [mat !! (i + 9 * p) | p ← [0 .. 8]]
         blkPos = [0, 1, 2, 9, 10, 11, 18, 19, 20]
         blk r c = [mat !! (3 * c + 27 * r + p) | p ← blkPos]
     forM_ [0 .. 2] $ λi →
       forM_ [0 .. 2] $ λj → do
         allDiff $ row $ i + 3 * j
         allDiff $ col $ i + 3 * j
         allDiff $ blk i j
     return mat
```

listing 5.5. For example, the expression tree corresponding to the $a + b + c + d \mathbin{@\equiv} 711$ constraint is:

```
Equal
  (Add
    (Var a)
    (Add
      (Var b)
      (Add
        (Var c)
        (Var d)
      )
    )
  )
  (Const 711)
```

As the types different underlying solvers use to represent their constraint variables may differ, an abstract variable identifier is used instead of actual variables. This avoids the need for the expression type *Expr a* to be additionally parametrized in the type of the solver. This has the additional advantage of making the expression code useful beyond representing constraints.

As constraints in the high-level model are predicates, they are identical to Boolean expressions. This allows *reification*: models which use the truth value of one constraint to be used as argument to another constraint.

Code listing 5.5 *Expr a* definition

```
newtype VarId = VarId Int
data Expr a where
  Var    :: VarId → Expr a
  Const  :: Eq a ⇒ a → Expr a
  Plus   :: Expr Int → Expr Int → Expr Int
  Minus  :: Expr Int → Expr Int → Expr Int
  Mult   :: Expr Int → Expr Int → Expr Int
  Dom    :: Expr Int → (Expr Int, Expr Int) → Expr Bool
  Equal  :: Eq a ⇒ Expr a → Expr a → Expr Bool
  Less   :: Expr Int → Expr Int → Expr Bool
  Not    :: Expr Bool → Expr Bool
  Or     :: Expr Bool → Expr Bool → Expr Bool
  And    :: Expr Bool → Expr Bool → Expr Bool
  AllDiff :: [Expr Int] → Expr Bool
  Exists :: (Expr a → Expr Bool) → Expr Bool
type FDConstraint = Expr Bool
```

Because we want to use constraint variables and expressions over them interchangeably, the expression type itself is made instance of *Term s*:

```
getNextId :: FDSolver s ⇒ FDWrapper s VarId
instance Term (FDWrapper s) (Expr Int) where
  newvar = do
    nid ← getNextId
    return (Var nid)
```

Thus, requesting a new Boolean variable in the high-level model amounts to simply creating a new identifier which is returned wrapped in a *Var* constructor. *getNextId*'s implementation is not shown, but simply increments a counter in the *FDState s* state, and returns its old value.

Using a GADT for the expressions allows us to use Haskell's type system in the language defined by the expression trees.[1] An alternative way for doing such an embedding, which doesn't require GADT's, but is less convenient, is given in

[1] In practice, more type constraints are necessary to use this representation, as pattern matches on polymorphic constructors cannot assume a fixed data type for the values matched against. Either type classes defining the allowed data types are necessary, or alternatively, separate expression types, for example *ExprInt* and *ExprBool*, each with their own relevant constructors. For clarity, these are omitted in the text.

Code listing 5.6 Operators for writing expressions

```
a  +    b = a `Plus` b
a  -    b = a `Minus` b
a  *    b = a `Mult` b
a @≡    b = a `Equal` b
a @≠    b = Not (a `Equal` b)
a @<    b = a `Less` b
a @>    b = b `Less` a
a  @:   b = a `Dom` b
a @≥    b = (a + 1) @> b
allin a b = foldr And (Const True) [x @: b | x ← a]
¬       b = Not b
allDiff b = AllDiff b
```

(Carette, Kiselyov, and Shan 2009). For now, we limit ourselves to integers and Booleans — more data types will be added later.

For example,

```
a `Equal` b
```

requires a and b to be of the same type, but is itself of type *Expr Bool*.

On top of the core primitives, syntactic sugar and a number of convenient abstractions are provided. Standard arithmetic operators and integer literals can be used for *Expr Int* thanks to the following instance of Haskell's *Num* type class:

```
instance Num (Expr Int) where
  fromInteger = Const ∘ fromInteger
  a + b       = a `Plus` b
  a − b       = a `Minus` b
  a ∗ b       = a `Mult` b
```

Thus, *Plus x* (*Mult* (*Const* 2) *y*) is written succinctly as $x + 2 * y$.

A summary of @-prefixed operators — specific for writing expressions — is given in Code listing 5.6.

The constraint operators are actually more advanced than that. For instance, the first one is defined as:

```
a + b = simplify $ a `Plus` b
```

where *simplify* performs a number of simplifications on the passed expression, and is shown in Code listing 5.7.

Code listing 5.7 Expression simplification logic

```
simplify :: Expr a → Expr a
simplify (Plus (Const a) (Const b)) = Const $ a + b
simplify (Plus a (Const 0)) = a
simplify (Mult (Const 0) a) = Const 0
simplify (Plus (Const a) (Minus (Const b) c)) =
  simplify $ Minus (Const $ a + b) c
simplify ... = ...
simplify x = x
```

Thus + and the other operators act as smart constructors that first simplify their arguments. Argument simplification is a solver-independent model transformation built into our framework. It heuristically minimizes the number of variable occurrences and the number of nodes in the expression tree. Of course, programmers can define their own libraries of abstractions and model transformations in entirely the same style.

Many similar simplifications can be implemented, drawing from the wide range of mathematical identities between expressions. This includes propagation of constant expressions (like the example above), but also exploiting associativity and distributivity between operations, normalization to improve detection of identical expressions, and tautologies.

5.3.2 Expressions as search trees

Using the above directly introduces the necessity of calling *addC* for every single constraint — a small but inconvenient syntactic overhead that breaks the paradigm of writing the model as a series of Boolean expressions. For example, the inner part of the *for* construct in the Sudoku example would have to be written as

```
forM [0..2] $ λi →
  forM [0..2] $ λj → do
    addC ∘ Left $ allDiff $ row $ i + 3 * j
    addC ∘ Left $ allDiff $ col $ i + 3 * j
    addC ∘ Left $ allDiff $ blk i j
```

The trivial solution is to make a second version of all operations that create values of type *Expr Bool*, which implicitly call *addC* ∘ *Left*, letting it create values of type *Solver s* ⇒ *SearchTree s* () instead. Early implementations of FD-MCP made a distinction between expressions and constraints, for this reason.

One can however observe that Boolean expressions can be mapped to a restricted set of *SearchTree s* () values, and back. This allows avoiding duplicate expression constructs, and treat expressions and *SearchTree* actions identically.

To do so, we first introduce a dummy solver, which only exists for writing encapsulated expressions. Search trees over *DummySolver* are always mapped back to expressions before execution:

```
data DummySolver a = DummySolver
instance Monad DummySolver where
  return _ = DummySolver
  _ >>= _  = DummySolver
instance Solver DummySolver where
  type Constraint DummySolver = Either FDConstraint ()
  type Label DummySolver = ()
  run _ = error "Trying to run dummy solver"
```

The following functions map between *FDConstraint* (Boolean expressions) to and from *SearchTree DummySolver* ():

```
model2tree :: (Solver s, Constraint s ~ Either FDConstraint a) =>
              FDConstraint -> SearchTree s ()
model2tree = addC . Left


tree2model :: SearchTree DummySolver () -> FDConstraint
tree2model (Solution _) = Const True
tree2model Fail         = Const False
tree2model (Branch l)   = foldr (@∨) (Const False) $ map tree2model l
tree2model (Add c s)    = c @∧ tree2model s
tree2model (NewVar f)   = Exists (tree2model . f)
```

The $\sim$ constraint on *model2tree* represents type equality: it requires a type s such that *Constraint s* is equal to *Either FDConstraint a*, for some a. This allows it to produce *SearchTree DummySolver* () values as well as *SearchTree* (*FDWrapper s*) () values.

By applying *model2tree* to every operation that creates Boolean expressions, and *tree2model* to each of its Boolean expression arguments, we get the best of both worlds: expressions are usable as *SearchTree* actions immediately, and can still be used as argument to other operations that use Boolean expressions. Type inference causes all second and deeper level Boolean expressions to be of type *SearchTree DummySolver* (), invisible to the user. For example:

```
(@∧) :: (Solver s, Constraint s ~ Either FDConstraint a) =>
        SearchTree DummySolver () -> SearchTree DummySolver () ->
```

```
        SearchTree s ()
a @∧ b = model2tree $ simplify $ tree2model a `And` tree2model b
```

5.4 Translation

In this section, a straightforward translation scheme from the expression language to constraints for the underlying solver is presented. A more general and powerful scheme will be presented in Chapter 7.

5.4.1 Interface

Not every *Solver* can serve as a back-end for *FDWrapper*-wrapped *Solver*s. Some specific capabilities are necessary:

- Support for integer and Boolean variables
- (At least) arithmetic constraints
- (Preferably) support for global constraints
- Conversion routines to help the translation process

These are bundled in a type class, *FDSolver*. Back-ends are responsible for supplying the relevant instance.

```
class (Solver s, Term s (FDIntVar s), Term s (FDBoolVar s))
  ⇒ FDSolver s where
  type FDIntVar s :: ∗
  type FDBoolVar s :: ∗
  compile_constraint :: Expr Bool → FDModel s ()
  setFailed :: s ()
```

The *FDSolver* type class makes three demands of a solver *s*:

- It must provide types *FDIntVar s* and *FDBoolVar s* for its terms.
- The function *compile_constraint* must take care of converting from an individual FD-MCP constraint to a model for the solver. Note that, to allow mapping a single FD-MCP constraint to a conjunction of solver constraints involving auxiliary variables, this function returns a model rather than a single constraint. This model is not allowed to contain any disjunctions.
- An action to set the internal *Solver* state to failed, *setFailed*, must be provided. This is called in case the translation process itself already encounters a conflict, without producing any constraints that cause this failure.

The following law specifies the *compile_constraint* function:

Denotation Preservation The *compile_constraint* function preserves denotation iff

$$[\![\cdot]\!] \circ \mathit{compile\_constraint} \equiv [\![\cdot]\!]$$

where $[\![\cdot]\!]$ maps a model or constraint onto its denotation, i.e., the set of possible solution tuples for the constraint variables appearing in the expression. $\equiv$ denotes extensional equality: $[\![x]\!] \equiv [\![y]\!]$ iff the projections (as defined in Section 2.2) of $[\![x]\!]$ and $[\![y]\!]$ on the intersection of the variables of x and y are identical.

This is easily integrated in the wrapper code, by changing *FDWrapper*'s *Solver* instance definition to:

```
instance FDSolver s ⇒ Solver (FDWrapper s) where
  ...
```

As *FDSolver s* itself has *Solver s* as super-constraint; this subsumes the earlier *Solver s* ⇒ requirement.

The type of *compile_constraint* is *FDModel s* (). Instead of directly adding the translated constraints to the solver, they are returned as a search tree. This has the advantage of allowing deeper inspection of the result, and additional transformations before actual application on the solver state. It is in fact required for compilation to C++ code (see Chapter 6). As *Constraint* (*FDWrapper s*) can be both a high-level model expression or a low-level constraint, *compile_constraint* is supposed to reduce the expression to the low-level variant. However, high-level expressions can be created as well, which are passed recursively to *compile_constraint* again, implicitly doing a fix-point computation:

```
instance FDSolver s ⇒ Solver (FDWrapper s) where
  ...
  add (Left m)  = tryFD $ untree ∘ compile_constraint $ m
  add (Right c) = wrap (add c)



untree :: Solver s ⇒ SearchTree s a → s (Maybe a)
untree (Solution a)  = return $ Just a
untree Fail          = return Nothing
untree (Branch _)    = error "Branching not supported"
untree (Add c s)     = add c ≫ untree s
untree (Dynamic d) = d ≫= untree
untree (NewVar f)  = newvar ≫= (untree ∘ f)
tryFD :: FDSolver s ⇒ FDWrapper s (Maybe a) → FDWrapper s ()
tryFD a = do
```

```
res ← a
when (isNothing res) $ wrap setFailed
```

For the actual implementation of *add*, the result of *compile_constraint* is passed to *untree* to remove the *SearchTree* wrapping, and turning it in a structureless solver action. Failure is dealt with using *tryFD*, which turns a returned *Nothing* into a call to *setFailed* on the underlying *Solver*.

5.4.2 Defunctionalization

The data structures used in the front end are easy to construct and combine, but not optimal for further processing. Especially the fact that functions are stored directly (see *NewVar* and *Exist*) inside ADTs is problematic, as the only possible operation on function types is application. There is no way to inspect or pattern match against a function. Therefore, as a first step, we "defunctionalize" the data.

This means translating from the *Expr* datatype as defined above, to a similar one with identical constructors, except:

```
Exists :: (Expr a → Expr Bool) → Expr Bool
```

is replaced by

```
Exists' :: VarId → Expr Bool → Expr Bool
```

through a translation function that maps *Exists f* to *Exists' nId* (*f* (*Var nId*)), with *nId* equal to a newly generated *VarId*. This representation is less safe, as it allows references to variables outside of the scope where they are defined, but does allow pattern matching.

The whole *Expr*-supporting code does not need to be duplicated for this, though. A generalized *Expr* is possible:

```
data GExpr t a where
  Term  :: t a → GExpr t a
  Const :: Eq a ⇒ a → GExpr t a
  Plus  :: GExpr t Int → GExpr t Int → GExpr t Int
  ...   :: ...
```

Term has been generalized here to take anything of type *t a*, where *t* is a type parameter to GExpr. For the "functional" expressions, *t* is something that encodes *Var* and *Exist*, and for the defunctionalized version, *Var* and *Exist'*:

```
data TermF a where
  Var    :: VarId → TermF a
  Exists :: (ExprF a → ExprF Bool) → TermF Bool
```

```
data TermU a where
  Var' :: VarId → TermU a
  Exists' :: VarId → ExprU Bool → TermU Bool
type Expr  a = GExpr TermF a
type ExprU a = GExpr TermU a
```

For now, *Exists* is the only construct that introduces a new variable in the expression tree. In the next chapter more such constructs will be added. Concerning *Exists* itself, however, an optimization is possible through a mathematical identity:

$$\exists x \,:\, P(x) \wedge (\exists y \,:\, Q(x,y))$$
$$\Updownarrow$$
$$\exists x \, \exists y \,:\, P(x) \wedge Q(x,y)$$

This means that

```
Exists (λx → p x `And` (Exists (λy → q x y)))
```

or, equivalently:

```
Exists' x (p x `And` (Exists' y (q x y)))
```

could be rewritten as

```
Exists' x (Exists' y (p x `And` q x y))
```

Or, if we assume all variables to be implicitly existentially quantified, simply:

```
p x `And` q x y
```

This means we can do without *Exists'*, and just remove all *Exist* nodes when defunctionalizing:

```
data TermU a where
  Var' :: VarId → TermU a
defunction' :: Expr a → State Int (ExprU a)
defunction' (Term (Exists f)) = do
  num ← get
  modify (+1)
  defunction ∘ f ∘ Term ∘ Var' $ VarId num
defunction' (Const x) = pure $ Const x
defunction' (And a b) = And <$> defunction' a <*> defunction' b
```

```
...
defunction :: Int → Expr a → ExprU a
defunction num expr = runReader (defunction' expr) num
```

defunction′ works using a *State* monad here. In the state the current number of variables is stored. When processing an *Exists* constructor, it is removed by retrieving the number in the current state, increasing it by one, and then applying the defunctionalization recursively to the function applied to the new variable. To descend into constructors like *And*, the applicative functor operators <$> and <∗> are used (see Section 3.6.4 and Section 3.7.1).

The full defunctionalization is then performed by running the *State* monad — in which the *defunction′* operation runs — with the current number of introduced variables (through *newvar*, stored in *FDState s*) as initial state.

For the rest of this text, we will assume the expression tree is properly defunctionalized, with a resulting data type *Expr a*, and for brevity, without intermediate *Term* constructor.

5.4.3 Default compilation

For some expressions, a mathematically equivalent yet less structured form may exist. In particular for global constraints this is often possible.

Through the use of a fix-point computation, more structured expressions can be reduced to less structured ones as a default fallback compilation strategy. After each reduction step, the *compile_constraint* function is invoked again to attempt a *Solver*-specific compilation.

One way to implement this, is through the use of mixins (Bracha and Cook 1990):

```
type Mixin a = a → a
mixin :: Mixin a → a
mixin = fix
(<@>) :: Mixin a → Mixin a → Mixin a
(<@>) = (∘)
```

A mixin is a value (typically a function) that receives one additional argument: the fully combined function itself. This allows functions to access a *this* object while overloading operations, comparable to object-oriented languages' use thereof.

In the case of *compile_constraint*, instances can use an implementation like:

```
instance FDSolver MySolver where
  ...
  compile_constraint =
    mixin (compile_mysolver_specific <@> compile_general)
```

In which case the *compile_general* function is provided by the framework and shared between all instances.

```
compile_general :: FDSolver s ⇒ Mixin (FDConstraint → FDModel s ())
compile_general this (Const False) = Fail
compile_general this (Const True)  = return ()
compile_general this (AllDiff l)   = conj [ this (x @≠ y)
                                          | (x : ys) ← tails l, y ← ys
                                          ]
compile_general this x             = error $ "Cannot compile " ++ show x
```

This provides a default compilation strategy for *Const True* and *Const False*, and a fall-back reduction for *AllDiff*. This strategy allows the *Expr* data structure to evolve independently from the solvers. As long as a default reduction is provided, all solvers keep working without any code change. Note however that reducing a constraint to several smaller ones may significantly reduce the solving performance. As explained in Section 2.3.1, this may reduce the consistency level attained.

5.4.4 Variable creation

One other common operation is the introduction of new variables. The expression layer identifies variables using abstract *VarId*'s, but at some point these need to be mapped to real variables for the underlying solver.

When doing so, we do not want to create the same variable multiple times — if the *VarId* is identical, the same variable must be returned as well. This can be solved generically instead of per-*Solver*.

We store previously created variables inside *FDState s*, which already contained the number of created *VarId*'s as well:

```
data FDState s =
  FDState { nVarId :: Int
          , intVars :: Map (Expr Int) (FDIntVar s)
          , boolVars :: Map (Expr Bool) (FDBoolVar s)
          }
```

This is identical to

```
data FDState s = FDState Int
                         (Map (Expr Int) (FDIntVar s))
                         (Map (Expr Int) (FDBoolVar s))
```

but additionally defines the extraction functions $nVarId :: FDState\ s \to Int$, $intVars :: FDState\ s \to Map\ (Expr\ Int)\ (FDIntVar\ s)$ and $boolVars :: FDState\ s \to Map\ (Expr\ Bool)\ (FDBoolVar\ s)$. Furthermore, it allows constructions to be written using *record syntax*. For example:

```
initFDState :: FDState s
initFDState = FDState { nVarId = 0
                      , intVars = Map.empty
                      , boolVars = Map.empty
                      }
```

When an ADT constructor has many fields, it is often preferable to introduce explicit names for them.

Note that the variables are stored with the entire expression as key, and not just the *VarId*. This allows the mechanism to be used more generally: sometimes expression nodes that are not references to high-level variables need to be turned into low-level variables as well (auxiliary variables).

Utility functions can be provided that inspect this cache for a certain expression, and if found return it, and otherwise execute a passed action to create it.

```
cacheExprInt :: FDSolver s ⇒ Expr Int → FDModel s (FDIntVar s)
                           → FDModel s (FDIntVar s)
cacheExprInt expr act = do
  state ← liftTree ∘ FDWrapper $ get
  case Map.lookup expr (intVars state) of
    Nothing → do
      new ← act
      let newIntVars = Map.insert expr new $ intVars state
      liftTree ∘ FDWrapper $ put state { intVars = newIntVars }
      return new
    Just x → return x
```

Most commonly, it is used for the creation of new variables, leading to the shorthand:

```
cacheNewInt :: FDSolver s ⇒ Expr Int → FDModel s (FDIntVar s)
cacheNewInt expr = cacheExprInt expr $ liftTree ∘ wrap $ newvar
```

5.5 Labeling

In order to be actually usable as a solver, the *FDWrapper*-wrapped solver must expose access to its variables — allowing branching and querying of solutions. To this end, the *EnumTerm* type classes must be implemented.

However, no information is available about the domains of expressions — only about the underlying variables they are represented with. We reuse the caching system here: to retrieve information about an expression, it is looked up in the cache, and *EnumTerm* calls are redirected to the resulting variable.

Hence, the only implementation change necessary is the addition of two instances:

```
instance (FDSolver s, EnumTerm s (FDIntVar s)) ⇒
         EnumTerm (FDWrapper s) (Expr Int) where
  ...
instance (FDSolver s, EnumTerm s (FDBoolVar s)) ⇒
         EnumTerm (FDWrapper s) (Expr Bool) where
  ...
```

We require from implementations that each variable which is called into existence directly using *exists* and occurs in one or more constraints, to be present in the cache. This is easy to do: *cacheExprInt* or *cacheNewInt* must be called for each *Var* constructor encountered. This guarantees that all variables of the high-level model can be branched over.

However, it does not guarantee the presence of other expressions in the cache — in a constraint model with variables x and y it may not be possible to branch over $x + y$. This restriction will be lifted in Chapter 7.

5.6 Back-ends

This section explains how to implement actual *Solver* back-ends for FD-MCP, by discussing two implementations: a simple proof-of-concept Haskell-only solver, and a binding to the Gecode C++ library.

5.6.1 Overton solver

The Overton FD solver is a basic finite domain solver in Haskell. It has been adapted from an implementation by David Overton[2] to instantiate the FD-MCP framework.

Firstly, a *Solver* instance incorporates the Overton FD solver in the general MCP framework:

```
instance Solver OvertonFD where
  type Constraint OvertonFD = OConstraint
  add c                     = addFD c
  run p                     = runFD p
instance Term OvertonFD FDVar where
  newvar = ...
```

where *addFD* and *runFD* are functions provided by the solver itself, *FDVar* is the type of integer terms, and *OConstraint* represents the supported constraints:

[2] http://overtond.blogspot.com/2008/07/pre.html

```
data OConstraint =
    OHasValue FDVar Int
  | OSame     FDVar FDVar
  | ODiff     FDVar FDVar
  | OLess     FDVar FDVar
  | OAdd      FDVar FDVar FDVar
  | OSub      FDVar FDVar FDVar
  | OMult     FDVar FDVar FDVar
```

With the following denotational rules:

$$
\begin{aligned}
[\![OHasValue\ x\ v]\!] &\equiv [\![x]\!] = v \\
[\![OSame\ x\ y]\!] &\equiv [\![x]\!] = [\![y]\!] \\
[\![ODiff\ x\ y]\!] &\equiv [\![x]\!] \neq [\![y]\!] \\
[\![OLess\ x\ y]\!] &\equiv [\![x]\!] < [\![y]\!] \\
[\![OAdd\ x\ y\ z]\!] &\equiv [\![x]\!] + [\![y]\!] = [\![z]\!] \\
[\![OSub\ x\ y\ z]\!] &\equiv [\![x]\!] - [\![y]\!] = [\![z]\!] \\
[\![OMult\ x\ y\ z]\!] &\equiv [\![x]\!] * [\![y]\!] = [\![z]\!]
\end{aligned}
$$

The *FDSolver* instance also enables the FD modeling language. Note that there is only one type of variable, *FDVar*, so that is used for both integer and Boolean variables.

```
instance FDSolver OvertonFD where
  type FDIntVar OvertonFD = FDVar
  type FDBoolVar OvertonFD = FDVar
  compile_constraint = mixin (convert <@> compile_general)
```

The *OvertonFD* solver's term language is much more restricted than that of the generic FD-MCP modeling language: it only provides single variables (*FDVar*), compared to the more extensive *Expr*-expressions. The translation function *convert* needs to overcome this syntactic mismatch.

```
convert :: Mixin (FDConstraint → FDModel OvertonFD ())
convert self (Less a b) = do va ← decomp a
                             vb ← decomp b
                             addC $ Right $ OLess va vb
convert self ...        = ...
convert self expr       = self expr
```

Since no constraints of a higher level than simple relations between *FDVar*s are available, every subexpression needs to be mapped to a separate variable first. This

is what *decomp* is for, producing an *FDVar* from an expression, with all constraints to connect it to the variables corresponding to its subexpressions already in place.

```
createvar :: FDSolver s ⇒ FDModel s (FDIntVar s)
createvar = liftTree ∘ wrap $ newvar

decomp :: Expr Int → FDModel OvertonFD FDVar
decomp expr@(Var _) = cacheNewInt expr
decomp expr@(Const i) = do
  var ← createvar
  addC ∘ Right $ OHasValue var i
  return var
decomp expr@(Plus a b) = do
  va ← decomp a
  vb ← decomp b
  vn ← createvar
  addC ∘ Right $ OAdd va vb vn
  return vn
```

The function uses *cacheNewInt* to instantiate variables, while preventing double creation. For the translation of *Const i* and *Plus a b*, a new auxiliary variable is introduced using *newvar* (wrapped and lifted as *createvar*), and a constraint is posted on it using *addC*. This allows using the compact modeling language of the FD framework, although the *OvertonFD* solver only provides a much more primitive language.

Note that *compile_constraint* conforms to the law given in Definition 5.4.1. For example, assume an input expression *Plus a b*. Its denotation is $[\![a]\!]+[\![b]\!]$. Running *decomp* on it, results in an output expression v combined with the additional constraint *OAdd va vb v*, where *va* and *vb* are the result of applying *decomp* on a and b recursively. Assuming by induction that $[\![va]\!] \equiv [\![a]\!]$ and $[\![vb]\!] \equiv [\![b]\!]$, $[\![OAdd\ va\ vb\ v]\!] \equiv [\![v]\!] = [\![va]\!] + [\![vb]\!]$ leads to $[\![v]\!] \equiv [\![a]\!] + [\![b]\!]$.

While this approach is very inefficient, it supports arbitrarily complex expressions. Furthermore, it is the only possibility when the underlying solver does not support more complex constraints.

Example The tree corresponding to the sum constraint from the grocery problem is shown in Code listing 5.8. Both constraints are mapped to a combination of *NewVar* and *Add* nodes. All subexpressions in the original FD model have been decomposed into an introduction of a new variable and an additional constraint. The variables a, b, c, d refer to the original constraint variables, while i_1, i_2, i_3 are auxiliary variables, introduced by the translation.

While it is obvious that prevention of duplicate creation of variables is necessary for expression-level variables, there is no harm in doing so for each and every

Code listing 5.8 Grocery example: model tree for sum constraint

```
NewVar $ λi1 →
  Add (OAdd a b i1) $
    NewVar $ λi2 →
      Add (OAdd i1 c i2) $
        NewVar $ λi3 →
          Add (OAdd i2 d i3) $
            Add (OHasValue i3 711)
              Solution ()
```

subexpression encountered while executing *decomp*. This would mean that for example when the subexpression $a + b$ occurs twice in an expression, and it is decomposed into a new variable v with constraint *OAdd a b v* the first time, the second time v is simply reused immediately. This is essentially a simple way for implementing *common subexpression elimination* (CSE). Section 7.3.1 will extend the effectivity of CSE a lot further.

We can implement this using *cacheExprInt*:

```
decomp' :: Expr Int → FDModel OvertonFD FDVar
decomp' expr = cacheExprInt expr $ help expr
  where help :: Expr Int → FDModel OvertonFD FDVar
        help (Var _) = createvar
        help (Const i) = do vn ← createvar
                            addC ∘ Right $ OHasValue vn i
                            return vn
        help (Plus a b) = do vn ← createvar
                             va ← decomp a
                             vb ← decomp b
                             addC ∘ Right $ OAdd va vb vn
                             return vn
```

So, each request to decompose is initially redirected to a cache lookup using *cacheExprInt*, and if this fails, the actual decomposition is done using an auxiliary function *help*. *help* then directly invokes *createvar* when necessary.

5.6.2 Gecode solver

Gecode (Gecode Team 2006) is a particularly attractive target for modeling languages: it is an efficient, open, free and portable constraint solving library for C++. At the same time, the imperative and low-level nature of C++ makes Gecode more

suitable as a back-end for a modeling language than a direct implementation language. It provides a predefined set of variable types, constraints, propagators and search strategies. Furthermore, it allows users to implement their own, without overhead compared to the built-in ones.

Rather than to run an FD-MCP model on a solver in Haskell, this section shows how to run it on a solver backed by Gecode in C++. This approach illustrates three practical advantages of FD-MCP. Firstly, the framework is able to reuse existing FD solvers implemented in other programming languages. Secondly, this approach benefits from the performance characteristics of a C++ implementation. Thirdly, the high-level modeling language of the framework becomes available for low-level solvers.

The *Gecode* solver is a Haskell abstraction of the actual Gecode C++ solver. It defines the syntax of constraints and terms understood by Gecode:

```
instance Term Gecode IntVar where ...
instance Term Gecode BoolVar where ...

instance Solver Gecode where
  type Constraint Gecode = GConstraint
```

As many Gecode constraints support arguments that are either variables or constants, we introduce an auxiliary type that can represent either:

```
data IntArg =
    IAVar IntVar   -- reference to variable
  | IAConst Int    -- constant integer
```

The language of constraints is richer than that of the Haskell constraint solver:

```
data GConstraint =
    CAbs IntArg IntArg
  | CDiv IntArg IntArg IntArg
  | CMod IntArg IntArg IntArg
  | CRel IntArg GOper IntArg
  | CDom IntArg Int Int
  | CMult IntArg IntArg IntArg
  | CAllDiff [IntVar]
  | CLinear [(IntVar, Int)] GOper Int
data GOper =
    GEqual
  | GDiff
  | GLess
```

Again, the denotations follow the expected pattern:

$$
\begin{aligned}
[\![CAbs\ x\ y]\!] &\equiv |[\![x]\!]| = [\![y]\!] \\
[\![CDiv\ x\ y\ z]\!] &\equiv \frac{[\![x]\!]}{[\![y]\!]} = [\![z]\!] \\
[\![CMod\ x\ y\ z]\!] &\equiv [\![x]\!]\ mod\ [\![y]\!] = [\![z]\!] \\
[\![CRel\ x\ (\oplus)\ y]\!] &\equiv [\![x]\!][\![\oplus]\!][\![y]\!] \\
[\![CDom\ x\ l\ h]\!] &\equiv [\![x]\!] \in \{l, \ldots, h\} \\
[\![CMult\ x\ y\ z]\!] &\equiv [\![x]\!][\![y]\!] = [\![z]\!] \\
[\![CAllDiff\ l]\!] &\equiv \bigwedge_{i \neq j} [\![l_i]\!] \neq [\![l_j]\!] \\
[\![CLinear\ l\ (\oplus)\ c]\!] &\equiv \sum_i l_{i,b} * [\![l_{i,a}]\!]\ [\![\oplus]\!]\ c
\end{aligned}
$$

$$
\begin{aligned}
[\![GEqual]\!] &\equiv = \\
[\![GDiff]\!] &\equiv \neq \\
[\![GLess]\!] &\equiv <
\end{aligned}
$$

Note in particular the support for linear constraints *CLinear* and the all-different global constraint *CAllDiff*.

Naive translation We can simply reuse the decompositioning mechanism described in the previous section, creating a separate variable for each subexpression, and possibly reusing it when the same subexpression occurs again. This however fails to take advantage of two important possibilities Gecode offers:

1. Support for constraints that refer to constant values, instead of variables
2. Support for general linear constraints, over arbitrary amounts of variables at once

For example, with the naive compilation scheme, the constraint $x+y+2*z \geqslant 3$ would be decomposed into:

$$
\begin{aligned}
&v_1 \equiv x + y \\
&v_2 \equiv 2 * z \\
&v_3 \equiv v_1 + v_2 \\
&v_4 \equiv 3 \\
&v_3 \geqslant v_4
\end{aligned}
$$

each of which corresponds to an elementary constraint (*CLinear*, *CMult*, *CLinear*, *OHasValue*, *CLinear*). However, this whole constraint can easily be mapped to a single *CLinear* constraint.

Code listing 5.9 Decomposing functions for Gecode

```
decompInt :: Expr Int → Maybe (FDModel GecodeFD Int)
decompInt (Const a) = Just $ return a
decompInt _         = Nothing

decompVar :: Expr Int → FDModel GecodeFD IntVar
decompVar expr = cacheExprInt expr $ help expr
  where help :: Expr Int → FDModel GecodeFD IntVar
        help (Const a) = do vn ← createvar
                            addC ∘ Right $ CLinear [(vn, 1)] GEqual a
                            return vn
        help (Mult a b) = do vn ← createvar
                             va ← decompArg a
                             vb ← decompArg b
                             addC ∘ Right $ CMult va vb (IAVar vn)
                             return vn

decompArg :: Expr Int → FDModel GecodeFD IntArg
decompArg expr = maybe fallback useconst $ decompInt expr
  where fallback = decompVar expr >>= return ∘ IAVar
        useconst = (>>=return ∘ IAConst)
```

Specific translation To support efficient translation without excessive decompositioning, we use a more elaborate scheme for Gecode. First of all, as the constraints have arguments that are either *IntVar*, *Int* or *IntArg*s, Code listing 5.9 lists three decompositioning functions. The first one, *decompInt* returns a result wrapped in *Maybe*, as not every expression can be represented as a constant integer. The second one, *decompVar*, is the standard decomposer that always creates a new variable. Its logic for dealing with the *Mult* constructor recursively calls *decompArg*, the third function. This one tries to decompose using *decompInt*, and falls back to using *decompVar* otherwise.

To deal with linear constraints, we need support for linear (sub)expressions, however. First, a helper type is introduced:

```
data Linear = Linear [(IntVar, Int)] Int
```

whose denotation is

$$[\![Linear\ l\ c]\!] = \sum_i l_{i,b} * [\![l_{i,a}]\!] + c$$

A straightforward instance for *Num Linear* is possible, allowing these con-

structs to be added, subtracted and converted from integer constants. For multiplication, a separate function is necessary, as the product of two linear expressions is not necessarily again linear — one of them must be constant for this to be the case:

```
multLin :: Linear → Linear → Maybe Linear
multLin (Linear [] c) (Linear a b) =
  Just $ Linear (map (λ(x, y) → (x, c * y)) a) $ b * c
multLin l             (Linear [] c) = multLin (Linear [] c) l
multLin _             _             = Nothing
```

Internally, these operations use a simplification routine, to normalize and simplify them:

```
simplifyLin :: Linear → Linear
simplifyLin (Linear l c) = Linear (map process grouped) c
  where grouped           = groupBy (comparing fst) l
        process ((a, b) : r) = (a, b + sum (map snd) r)
```

This code first groups all terms of the linear expression per variable, and then adds their coefficients together per group.

Armed with this data structure and its supporting operations, a fourth decomposer can be written, shown in Code listing 5.10.

Furthermore, *decompVar* can reuse this logic:

```
decompVar :: Expr Int → FDModel GecodeFD IntVar
decompVar ... = ...
decompVar expr@(Plus _ _)  = decompLin expr >>= decompVarFromLin
decompVar expr@(Minus _ _) = decompLin expr >>= decompVarFromLin
```

The result is four interdependent decomposers which build descriptions of expression nodes, that try to remain constant or linear as long as possible, and fall back to auxiliary variables and constraints only when necessary. These are easily used to write the final *convertGecode* function, given in Code listing 5.11. Conversion of the *Less* expression constructor uses *decompLin*. It is based on the mathematical identity:

$$\begin{gathered}\sum_i x_{i,b} * [\![l_{i,a}]\!] + x_c < \sum_j y_{j,b} * [\![l_{j,a}]\!] + y_c \\ \Updownarrow \\ \sum_i x_{i,b} * [\![l_{i,a}]\!] - \sum_j y_{j,b} * [\![l_{j,a}]\!] < y_c - x_c\end{gathered}$$

Code listing 5.10 Decomposer for linear expressions

```
decompLin :: Expr Int → FDModel GecodeFD Linear
decompLin (Const a)   = return $ Linear [] a
decompLin (Plus a b)  = (+) <$> decompLin a <*> decompLin b
decompLin (Minus a b) = (−) <$> decompLin a <*> decompLin b
decompLin (Mult a b)  = do
  va ← decompLin a
  vb ← decompLin b
  case multLin va vb of
    Nothing → do
      vn ← createvar
      dva ← decompVarFromLin va
      dvb ← decompVarFromLin vb
      addC ∘ Right $ CMult (IAVar dva) (IAVar dvb) (IAVar vn)
      return $ Linear [(vn, 1)] 0
    Just l → return l
decompLin expr = do
  vx ← decompVar expr
  return $ Linear [(vx, 1)] 0
decompVarFromLin :: Linear → FDModel GecodeFD IntVar
decompVarFromLin (Linear l c) = do
  vn ← createvar
  addC ∘ Right $ CLinear l GEqual $ −c
  return vn
```

Code listing 5.11 Conversion routine for *Gecode* constraints

```
convertGecode :: Mixin (FDConstraint → FDModel GecodeFD ())
convertGecode self (Less a b) = do
  va ← decompLin a
  vb ← decompLin b
  case (va − vb) of
    Linear l c → addC $ Right $ CLinear l GLess (−c)
convertGecode self ... = ...
convertGecode self expr = self expr
```

Runtime The only remaining problem is how to use the generated *GConstraint* values to direct an actual solver in C++, that is: what do the *GecodeFD* solver, and its *run* and *add* functions look like?

To interact with C++, we use the Haskell Foreign Function Interface (FFI). This allows us to declare certain functions as *foreign*, and implemented in a C library instead of in Haskell. We provide a *glue* layer, partially implemented in Haskell and partially in C, which uses the FFI to let the two communicate.

Gecode internally uses a data type called a *space* to represent the state of a constraint solver. Spaces can be cloned to support branching, implicitly forming a search tree being explored. Constraints are posted to spaces and queued for propagation. When necessary, the queue can be processed, causing the space to go to a solved state, to a failed state, or to an unsolved state in which variable domains are constrained.

On the Haskell side, we represent these spaces using:

newtype *Space* = *Space* (*Ptr Space*)

The *Ptr* type is provided by the FFI system itself, and corresponds to a pointer type in C. This defines *GecodeModel* as representationally identical to such a pointer. By using a recursive type here, we can guarantee that no such value can every be created from Haskell — it is effectively a black box type, only created and inspected from the C side[3]. Note that it is not actually a pointer to another *Space*.

Some basic interaction operations are defined using the FFI:

foreign import `"space_create"` *spaceCreate* :: *IO Space*
foreign import `"space_destroy"` *spaceDestroy* :: *Space* → *IO* ()
foreign import `"space_clone"` *spaceClone* :: *Space* → *IO Space*

These define the Haskell values *spaceCreate*, *spaceDestroy* and *spaceClone* to be implemented by the C functions `space_create`, `space_destroy` and `space_clone` respectively. Note that as these operations describe interactions that fall outside of Haskell's referentially transparent world view, they produce values of type *IO* α.

Other operations provided by the glue layer include:

- *spaceFail* :: *Space* → *IO* (): put a space in a failed state
- *spacePropagate* :: *Space* → *IO Bool*: perform propagation and return whether the space is failed
- *postConstraint* :: *Space* → *GConstraint* → *IO* (): post a constraint in a space

[3]Note that we say C here, because that is what lies beneath the FFI. This C code itself is a tiny wrapper that itself calls the C++ functions from Gecode for its actual work.

- *newIntVar* :: *Space* → *IO IntVar*: create a new integer variable in the reference *Space*, and return its identifier (analogous for *newBoolVar*).

IntVar in the above functions is another black-box type, as it describes an identifier which is only used by the C side. We model it as an integer [4]:

```
newtype IntVar = IntVar VarId
```

As the FFI functions are *IO* actions, they need to be called from "within" an *IO* monad. The obvious choice is to make *GecodeFD* a wrapped *IO* action that carries a *Space* reference in its state:

```
newtype GecodeFD a = GecodeFD (State_T Space (IO a))
```

with the following *Solver* instance:

```
instance Solver GecodeFD where
  type Constraint GecodeFD = GConstraint
  type Label GecodeFD = Space
  add c = GecodeFD (lift $ postConstraint c)
  run = runGecode
  mark = GecodeFD (get >>= lift spaceClone)
  goto s = GecodeFD ((lift $ spaceClone s) >>= put)


instance Term GecodeFD IntVar where
  newvar = GecodeFD (lift newIntVar)
```

Requests to add constraints simply cause a lifted call to *postConstraint*. For disjunctive models, and branching in general, we use the copying technique in Gecode. Thus for the label of a solver state, we simply use the solver state, i.e., the Gecode space itself. Whenever creating a branch starting from a given space, we install a copy of that space as the current space so as not to affect other branches[5].

The only remaining part is the *runGecode* function, which needs to be of type

```
runGecode :: GecodeFD a → a
```

which is actually (removing the type-level abstraction provided by **newtype**):

```
runGecode :: State_T Space (IO a) → a
```

[4] In practice, the C type `int` corresponds to the Haskell type *CInt*, which we make abstraction from here. Similarly, values of type *GConstraint* are not passed directly to the C side, but processed and pattern matched against in the Haskell code, followed by a call to one of many C functions for posting particular constraints.

[5] Several optimizations are possible here, to prevent the excessive copying of spaces.

However, as explained in Section 3.6.2, it is not possible to remove the *IO* type. Doing so would break referential transparency, and imply execution of the encapsulated action upon evaluation. We can, however, guarantee that the complete action formed by binding solving actions together is always safe to execute *in its entirety*.

For such cases, a function is provided by Haskell to "break out of the *IO* jail": $unsafePerformIO :: IO\ \alpha \to \alpha$. This function indeed breaks the nice pure semantics of Haskell, and allows arbitrary side effects. It is up to the programmer to make sure no *observable* side effects are possible when using it.

Using *unsafePerformIO*, we can write:

```
runGecode :: GecodeFD a → a
runGecode (GecodeFD s) = unsafePerformIO $ do
  rootSpace ← spaceCreate
  ret ← evalState_T s rootSpace
  spaceDestroy rootSpace
  return ret
```

where *spaceCreate* is invoked to create a pristine root space. This root space is used as initial state for the execution of the encapsulated $State_T\ Space$. The root space can then be destroyed, and the result returned. Combined, this forms an action of type *IO a*, which is passed to *unsafePerformIO*.

Fixed search The Gecode back-end of the framework offloads constraint propagation on the Gecode solver, but still allows the programmer to program and specify the search heuristics through the high-level interface. We call this approach the *programmed search mode*. It has clear advantages in terms of flexibility (see Part III), but it does incur an interpretative penalty for search, which for many constraint problems has a considerable impact on the overall solving time.

In order to avoid the interpretative overhead for search, we provide a second mode of on-line use, the *fixed search mode*. This mode provides a fixed search strategy implemented in C++ for the Gecode solver. In this mode, labeling the model does not produce a whole subtree that is affected by the framework's search heuristics. Instead, a single node is generated on the MCP side that corresponds to many nodes in the Gecode solver which are processed by a fixed search strategy.

Implementation-wise, this boils down to providing an alternative *GecodeFD* type, which reuses all the normal *Solver* and *FDSolver* logic, but has a different *EnumTerm* instance for its variables.

5.7 Conclusion

Compared to the system shown in Section 4.2, the situation has improved:

- **Front-end layer**: Models can be written using a solver-independent high-level syntax, instead of manually writing constraints for the particular solver used.
- **Back-end solvers**: Several backends are added that are more powerful than the Haskell solver.
- **Intermediate layer**: Common operations for all FD-capable solvers can be implemented once and for all, allowing easy addition of new solvers.

The syntax itself allows models to be described as a series of Boolean expressions that refer to constraint variables. As in MCP itself, this is implemented using a few core primitives with syntactic sugar and utility function on top. Evaluation leads to an abstract syntax tree, on which simplifications are applied through pattern matching. Through the use of first-class functions, we can encode higher-order constructs in the expression tree. Finally, back-ends provide their own solver-specific translation routines that decompose the expression trees into underlying constraints. In particular, a Gecode-based back-end provides state-of-the-art constraint solving. Benchmarks for the FD-MCP system will be given in Section 7.5, after we've discussed further improvements to the translation system in Chapter 7.

Chapter 6

Off-line solving

6.1 Introduction

Instead of directly solving the model at the time a Haskell program containing the problem is executed, it is possible to generate a C++ program that will perform the actual solving at a later time. We call this the *off-line* mode of operation, contrary to the *on-line* mode which was used for the *OvertonFD* and *GecodeFD* solvers. There the constraint model is processed by the MCP framework, in collaboration with the solver, to produce solutions during execution of the Haskell program.

In contrast, the off-line mode concerns *staged compilation*: in the first stage, the FD model is processed by the MCP framework that produces code for the second stage in the solver's programming language; the stage-2 code produces solutions. The off-line mode was used in the original Gecode back-end of FD-MCP (Wuille and Schrijvers 2009b).

The off-line mode has a clear appeal for performance reasons: it avoids the interpretative overhead when solving the constraint model in the second stage. Of course, there is the compilation overhead of the first stage. We come back to this issue in the next section, where we considerably improve the usefulness of the off-line mode.

The on-line mode is more convenient: it allows immediate use of the solver results, supports model transformations, and allows programming the search. In contrast, the off-line Gecode solver provides a fixed search strategy – which is confounded by the fact that the FD solver is implemented in Haskell itself.

6.2 Integration

We have implemented an off-line mode for the Gecode solver: C++ code is generated that itself links with the Gecode library. A significant part of the logic

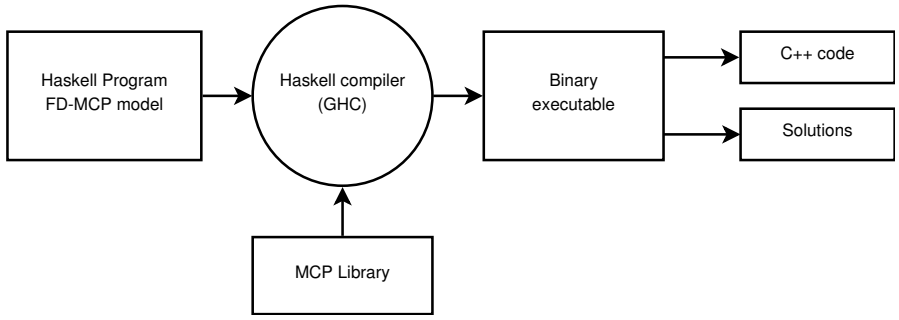

Figure 6.1: Offline solving architecture

implemented in the previous chapter can be reused, including the *FDSolver* instance and conversion functions.

However, instead of posting the constraints using the FFI, this *Solver* simply stores all its posted *GConstraint*s for later conversion to C++:

```
newtype OfflineGecodeFD a = OfflineGecodeFD (State OfflineState a)
  deriving (Monad, State_M OfflineState)
type OfflineState = [GConstraint]
instance Solver OfflineGecodeFD where
  type Constraint OfflineGecodeFD = GConstraint
  add c = modify (c:)
  run = ...
```

Obviously, no useful *EnumTerm* instance can be given for its variables. Nor do we need one: one cannot expect to use the search infrastructure for this solver, and find actual solutions.

Instead, *untree* from Section 5.4.1 is used to remove the tree structure from the model:

```
untree :: SearchTree s a → s (Maybe a)
```

When this succeeds (doesn't return *Nothing*), the state of the solver can be inspected to extract the constraints. The constraints and variables constituting the solution are passed to a code generator, which produces C++ code in the form of an Abstract Syntax Tree (AST). We don't go into implementation details, but represent pieces of code by values of the data type *C++*. Finally, a renderer converts this to nicely formatted C++ code:

```
codegen :: [GConstraint] → ([IntVar], [BoolVar]) → C++
```

Code listing 6.1 Parametrized model of the N queens problem

```
nqueens :: Int → FDModel s [Expr Int]
nqueens n = do                          -- define model function 'nqueens'
  q ← exist n                           -- new list 'q' of size n
  q `allin` (1, n)                      -- all variables in range 1..n
  foreach (1, n) $ λi →                 -- for i in 1..n
    foreach (i + 1, n) $ λj → do        -- for j in i+1..n
      q ! i     @≠ q ! j                --
      q ! i + i @≠ q ! j + j            -- constraints
      q ! i - i @≠ q ! j - j            --
  return q                              -- return result list
```

```
render :: C++ → String
```

As models in FD-MCP are written as values of type *FDModel s a*, we also need to remove the FD wrapping. The following function combines everything, and turns a *FDModel s* [*Expr Int*] whose return value represents the variables of the solution, to a *String* of formatted C++ code:

```
compile :: FDModel OfflineGecodeFD [Expr Int] → Maybe String
compile s = run $ untree $ render $ do
  exprs ← s
  vars ← mapM (λx → cacheExprInt x $ error "no variable found")
              exprs
  cons ← liftTree $ wrap get
  return $ codegen cons (vars, [])
```

An action is constructed that first runs the given model search tree, capturing its return expressions. These are then mapped to variables via *cacheExprInt*, producing an error if they are not found in the cache. The state of the solver, which contains the constraints, is then inspected using a lifted *get*. Both are then passed to *codegen* and *render*, and returned.

6.3 Parametrized models

Many FD models are naturally parametrized in a problem size and/or other instance-specific integer values. For instance, the n-queens problem is parametrized in the board size, the Golomb ruler problem is parametrized in the ruler size, ...

Code listing 6.1 shows a parametrized model of the N queens problem.

Such parametrization does not pose any problem for the on-line solvers. The parametrized model is simply written as a *model function* from one (or more) integer values to an FD model.

Instead of a regular

$$model :: SearchTree\ s\ a$$
$$model = \ldots$$

you write

$$pmodel :: Int \rightarrow SearchTree\ s\ a$$
$$pmodel\ n = \ldots$$

In order to solve the model, the model function is applied to the appropriate values, and the resulting model is handed to the on-line solver. No surprises. In effect, a model function defines a *problem class*, while the resulting model defines a *problem instance*. This introduces a separation between code and data.

For off-line solvers, we could follow the same technique. However, then we would obtain a non-parametrized off-line executable. Each time we would like to change the parameters, we would have to generate a new off-line executable! That is very costly in terms of compilation times, compared to the on-line solvers. The latter require only one invocation of the Haskell compiler for a parametrized model, while the former requires one invocation of the Haskell compiler and subsequently, for each instantiation of the parameters, an invocation of the C++ compiler. Moreover, the size of the off-line code is dependent on the problem size, because the framework fully flattens the model before generating code. Hence, the larger the problem size, the bigger the generated C++ code, and the longer the C++ compilation times. In summary, a new approach is necessary to make parametrized models practical for off-line solving.

The remainder of this chapter shows our approach for representing and compiling parametrized models. It has the two desirable properties:

- a parametrized model requires only a single invocation of the C++ compiler, and
- the generated code does not depend on the parameter value.

6.3.1 Parameters

We still represent parametrized models by model functions, but the functions take expressions rather than integers as arguments.

$$pmodel :: Expr\ Int \rightarrow SearchTree\ s\ a$$
$$pmodel\ n = \ldots$$

We still retain the above functionality for on-line solvers, as integer values can be lifted to FD expressions using the $Const :: Int \to Expr\ Int$ constructor. Moreover, $Expr\ Int$ is also an instance of the *Num* type class, so integer literals can be supplied directly as arguments: *pmodel* 1425.

Of more interest is of course the treatment of model functions for off-line solving. A model function is compiled by applying it to special $Expr\ Int$ values that represent *deferred values*. These deferred values are not known until the C++ stage. We denote a deferred value in the first stage as `$p$, where $p$ is the corresponding representation, a C++ `int` variable, in the second stage.

So using these deferred variables, we again obtain a model that can be compiled much as before. Only the deferred values require special care. They are mapped to `int` instance variables of the generated C++ class that represents the Gecode constraint model. A new instance of the problem is created by instantiating an object of that class with the desired integer values for the parameters.

To implement this, we turn back to the defunctionalization mechanism presented in Section 5.4.2. First, a new constructor is added to *Expr*:

newtype $ParId = ParId\ Int$
data $Expr\ a$ **where**
  ...
  $Param :: ParId \to Expr\ a$

Integration As the function is not stored inside an expression construct, but explicit, this means that if we want to support arbitrary amounts of arguments, different types need to be supported:

- For $pmodel :: SearchTree\ s\ a$
  we simply call *pmodel*
- For $pmodel :: Expr\ Int \to SearchTree\ s\ a$
  we call $pmodel\ (Param\ (ParId\ 1))$
- For $pmodel :: Expr\ Int \to Expr\ Int \to SearchTree\ s\ a$
  we call $pmodel\ (Param\ (ParId\ 1))\ (Param\ (ParId\ 2))$
- ...

This is possible using a type class and flexible instances[1]. First, we factor out the code in *compile* to extract the unstructured solver action in a type class:

type $OfflineGecodeWr = FDWrapper\ OfflineGecodeFD$

[1] A Haskell language extension that allows type class instances on more than simple type constructors.

```
class Compilable a where
  extract :: a → State_T Int OfflineGecodeWr ([GConstraint], [IntVar])
instance Compilable (FDModel OfflineGecodeFD [Expr Int]) where
  extract s = lift $ liftM fromJust $ untree $ do
    exprs ← s
    vars ← mapM (λx → cacheExprInt x $ error "not found") exprs
    cons ← liftTree $ wrap get
    return (cons, vars)
```

The *extract* function turns the search tree into a solver action that just computes the constraints and variables of the constraint problem. The solver type is transformed using $State_T$, to thread an integer as state with it. This integer carries the number of parameters introduced, and starts at zero.

Next, a recursive instance is defined to make function types also a member of this class:

```
instance Compilable a ⇒ Compilable (Expr Int → a) where
  extract f = do
    nPars ← get
    modify (+1)
    extract ∘ f $ Param $ ParId nPars
```

This code deals with functions by first requesting the value stored in the state, then increasing it by one, and finally recursively calling the *extract* function, with one level of abstraction removed.

All that is left now, is writing a generic *compile* function on top of this:

```
compile :: Compilable a ⇒ a → String
compile s = translate ∘ run $ evalState_T (extract s) 0
  where translate (cons, vars) = render $ codegen cons (vars, [])
```

This way, one can write

```
compile model
```

and get the compiled C++ code, independent of whether model takes one or more function arguments or not.

Translation Some changes to the translation process are necessary as well. An expression which does not contain any *Var* constructs, but still contains *Param* constructs, should — from the point of the generated C++ code — be considered constant. Even though it cannot be evaluated during the Haskell stage, this will be possible as soon as the deferred values are filled in.

Therefore, some logic from Section 5.6.2 requires modifications. In particular, we need a type to represent "constant" expressions. We can reuse the *GExpr* from Section 5.4.2 for this:

```
data NoTerm a = NoTerm
type CExpr a = GExpr NoTerm a
```

CExpr a now represents constant expressions that evaluate to type *a* at C++ run-time. By using an empty constructor as *Term*-type — which normally contains the *Var* constructor — we can guarantee the absence of variable references through the type system.

Thus, *CExpr* can be considered as analogous to *ExprU*, only even more restricted in that it not only requires defunctionalization, but complete absence of variables.

We write a general transformation function to convert between the different flavors of *GExpr*-based types. It resembles the *Traversable* pattern[2] (Mcbride and Paterson 2008), yet is of a higher order: the parameter to *GExpr* is not a full type, but a type function. Higher-kinded languages[3] allow abstraction from these differences, and could generalize *Traversable* to what we need here.

```
mapExpr :: Applicative f ⇒ (g a → f (h a)) → GExpr g a → f (GExpr h a)
mapExpr f (Term t) = Term <$> f t
mapExpr f (Const a) = pure $ Const a
mapExpr f (Plus a b) = Plus <$> mapExpr f a <*> mapExpr f b
mapExpr f ... = ...
mapNoVars :: GExpr g a → Maybe (GExpr h a)
mapNoVars = mapExpr $ const Nothing
```

mapNoVars turns all usage of *Term* constructors in the argument into *Nothing*. Applying using *TermF* as *g* and *Dummy* as *h*, we get

```
GExpr TermF a → Maybe (GExpr Dummy a)
```

or equivalently,

```
Expr a → Maybe (CExpr a)
```

Generalization using any *Applicative* instead of *Maybe* automatically extends its usage to other structures, like lists or any monad.

[2] Data structures that represent structured containers whose elements can be transformed individually, leaving the structure intact. Though not used in this text, *Expr* itself is a traversable.

[3] A "type of type" is called a *kind*. The kind of fully instantiated types (of which values can be created) is called ∗, while the kind of type functions like *IO* or *Maybe* is ∗ → ∗. *GExpr* has kind (∗ → ∗) → ∗ → ∗, as it takes a type function *t* and a result type *a* as arguments.

Armed with the *CExpr* type and its supporting conversion routine *mapNoVars*, the Gecode translation layer's *IntArg* can be adapted:

```
data IntArg =
    IAConst (CExpr Int)
  | IAVar    IntVar
```

Instead of using a simple integer as argument to *IAConst*, we use a full constant expression. Furthermore, instead of *decompInt* we turn to *decompConst*:

```
decompConst :: Expr Int → Maybe (FDModel GecodeFD (CExpr Int))
decompConst x = return <$> mapNoVars x
```

together with its straightforward integration in the other decomposers and conversion functions.

6.3.2 Indexed Constraint Variable Lists

This is not the end of the story, however. Parameters of type *Expr Int* have fewer uses than values of type *Int*. Indeed, the former can be used as arguments to constraints, but the latter can appear in many useful Haskell library functions as well as several functions of the MCP framework. Perhaps the most essential such function is *exist*, which creates a list of the specified number of constraint variables. In many parametric models, the number of constraint variables depends on the parameter value.

However, for the off-line solver, the integer value of the parameter is not available. Thus the actual creation of the list must be deferred from the on-line Haskell phase to the off-line phase. Moreover, we may wish to use a different data structure than a linked list in the off-line phase, such as an array in C++.

Hence, to allow writing models that can be used with both on-line and off-line solvers, we extend the expression component with list expressions:

```
data Expr t where
  ...
  List :: [Expr a] → Expr [a]
  Cat  :: Expr [a] → Expr [a] → Expr [a]
  Size :: Expr [a] → Expr Int
  At   :: Expr [a] → Expr Int → Expr a
```

This way, a list type is introduced in the expression language. Lists are constructed using the new *List* construct, which takes a list of expressions of the element type, or using the new *Cat* construct to concatenate two lists. Finally, a list of constant values is also possible using the earlier *Const* constructor. For example, *Const* $[1, 2, 3]$ is equivalent to *List* [*Const* 1, *Const* 2, *Const* 3].

Two new constructors represent inspection of a list: *Size* represents its length, and *At* represents an element at a particular position in it. The earlier *Equal* of type $Expr\ a \to Expr\ a \to Expr\ Bool$ is also applicable to list types, and represents list equality.

Furthermore, *FDSolver* instances require a new associated type *FDIntListType* together with a special creation function:

```
class FDSolver s where
  ...
  type FDIntListType s :: *
  fdNewIntList :: CExpr Int → s (FDIntListType s)
```

FDIntListType s doesn't need to be an instance of *Term s* even, as there is no requirement to be able to create a list of indeterminate size (as *newvar* would do, lacking a size argument).

For on-line solvers like *OvertonFD*, it is defined as an on-line Haskell list:

```
instance FDSolver OvertonFD where
  ...
  type FDIntListType OvertonFD = [FDIntType OvertonFD]
```

For off-line solvers like *OfflineGecodeFD*, a deferred list `c is used that only records an identifier *c* of the particular list:

```
newtype IntArr = IntArr VarID
instance FDSolver OfflineGecodeFD where
  type FDIntListType OfflineGecodeFD = IntArr
```

To support variable-size lists — where the number of elements depends on a deferred value — yet another *Expr* constructor is needed:

```
data Expr t where
  ...
  ExistN :: CExpr Int → (Expr [a] → Expr Bool) → Expr Bool
```

comparable to the *Exists* constructor for single variables. *ExistN* takes an additional argument — the list length — and requires its function argument to take a list type. Contrary to the *Exists* constructor however, this one does need a defunctionalized (see Section 5.4.2) version

```
ExistsN' :: CExpr Int → VarID → Expr Bool → Expr Bool
```

that cannot be considered implicit, as the knowledge about the list length would be erased. A default decomposer traversing such a constructor calls *fdNewIntList*,

and stores in the cache (see Section 5.4.4) a mapping from *Var nVarId* to the created *FDIntListType s* before recursing into decomposing its third argument.

On top of this, some syntactic sugar is provided to write search trees corresponding to expressions using the above constructors (see Section 5.3.2):

- *fdexist* :: *Expr Int* → *FDModel s* (*Expr* [*Int*]) creates a new list of specified size, and acts as a generalization of *exist*. The resulting *ExistN* constructor is mapped to normal *exist* calls for on-line solver, creates a new deferred list for off-line solvers. This function perform conversion to *CExpr Int* on its first argument, as the length of list must be known at creation time, and thus may fail. Note that the size of generated code for the latter is $O(1)$ (a single array declaration) as opposed to $O(n)$ like *exist*.

- (!) :: *Expr* [*a*] → *Expr Int* → *Expr a* returns an element at a given index in the list. The resulting *At* constructor is implemented in terms of Haskell list indexation (!!) for on-line solvers, but for off-line solvers a term denoting deferred indexation is returned. Then we have that $\langle\!\langle \grave{} c \,!\, i \rangle\!\rangle = c[\langle\!\langle i \rangle\!\rangle]$.

Global constraints form another class of functions that involve lists. These have been modified to support list expressions instead of Haskell lists:

- *allDiff* :: *Expr* [*Int*] → *FDModel s* () all expressions in the given list expression evaluate to mutually distinct values (ALLDIFFERENT constraint).

- *sorted* :: *Expr* [*Int*] → *FDModel s* () the given list expression is sorted (SORTED constraint).

- *allin* :: *Expr* [*Int*] → (*Expr Int*, *Expr Int*) → *FDModel s* ()
  all expressions in the given list expression have a value between the given lower and upper bounds.

6.3.3 Iteration

Often the above operations for lists are not expressive enough. Instead of imposing global constraints on a list or indexing specific entries, many models process all elements of a list one at a time. For this purpose an iteration construct is necessary.

Iteration Primitives

We introduce in our framework the iteration primitive *foreach*::(*Expr Int*, *Expr Int*) → (*Expr Int* → *FDConstraint*) → *FDModel s* (), whose denotation is:

$$[\![\mathit{foreach}\ (l, u)\ f]\!] \equiv \bigwedge_{i=l}^{u} [\![f\, i]\!]$$

For instance, we write $\bigwedge_{i=1}^{n}(c_i > i)$ as:

$$foreach\ (1, n)\ \$\ \lambda i \rightarrow (c\ !\ i)\ @{>}\ i$$

This is implemented by again extending the expression type:

$$\textbf{data}\ Expr\ a\ \textbf{where}$$
$$\ldots$$
$$ForEach :: (Expr\ Int, Expr\ Int) \rightarrow (Expr\ Int \rightarrow Expr\ Bool) \rightarrow Expr\ Bool$$

As with *Exists* and *ExistN* before, we use the fact that functions are first-class objects in Haskell, and store the inner expression as a function in the outer expression. This immediately adds support for nesting and reification.

For on-line solvers, *ForEach* is translated literally according to its semantics:

$$compile\ (ForEach\ (Const\ l, Const\ h)\ f)\ f) = conj\ [f\ i \mid i \leftarrow [l\,..\,u]]$$

However, for off-line solvers, the range of the loop may not be constant, if it depends on a model parameter. Even if we do know the range, we may choose not to flatten the loop if the range is too large. In these cases, *ForEach* is compiled to a C++ `for`-loop:

$$\langle\!\langle ForEach\ (l, u)\ f \rangle\!\rangle = \texttt{for (int i = } \langle\!\langle l \rangle\!\rangle \texttt{; i =< } \langle\!\langle u \rangle\!\rangle \texttt{; i++) \{ } \langle\!\langle f\ \texttt{`i} \rangle\!\rangle \texttt{ \}}$$

So the size of the generated code does not depend on the size of the iteration range.

Because iteration over the whole range, rather than a subrange, of a list occurs quite frequently, we introduce a second iteration construct $forall :: Expr\ [Int] \rightarrow (Expr\ Int \rightarrow FDConstraint) \rightarrow FDModel$, whose denotation is:

$$[\![forall\ c\ f]\!] \equiv \bigwedge_{v \in c} [\![f\ c]\!]$$

For instance, we write $\bigwedge_{v \in c}(v > i)$ as:

$$forall\ c\ \$\ \lambda v \rightarrow v\ @{>}\ i$$

or even shorter:

$$forall\ c\ (\ @{>}\ i)$$

forall is simply mapped to the more generic *foreach* construct:

$$forall\ c\ f = foreach\ (Const\ 1, Size\ c)\ \$\ f \circ (c!)$$

which means that we get the following C++ code:

$$\langle\!\langle forall\ \texttt{`}c\ f \rangle\!\rangle = \texttt{for (int i = 0; i < } \langle\!\langle size\ \texttt{`}c \rangle\!\rangle \texttt{; i++) \{ } \langle\!\langle f\ \texttt{`}c\texttt{[i]} \rangle\!\rangle \texttt{ \}}$$

Additional Higher-Order Constructs

While the above iteration primitives are expressive enough to formulate most list processing operations, often the formulation can be quite awkward and the resulting code rather inefficient (e.g., requiring many auxiliary lists). For that reason, we directly support additional higher-order list processing constructs besides $\forall\cdot$ and *foreach*.

Mimicking the standard Haskell functions *map* and *foldl*, we provide the following:

- *fdmap*:: $(Expr\ a \to Expr\ b) \to Expr\ [a] \to$ $Expr\ [b]$ transforms each element of a list using a specified function, similar to the standard Haskell function *map*.
- *fdfold*:: $(Expr\ a \to Expr\ b \to Expr\ a) \to Expr\ a \to$ $Expr\ [b] \to Expr\ a$ folds a list to a single expression, similar to the standard Haskell function *foldl*.

Again, the *Expr* constructors are extended with new constructors (*Map* and *Fold*), and simplifications are applied when creating them. For example, applying a *fdmap* to another *fdmap* results in a single *Map*, while applying an *fdfold* on an *fdmap* results in a single *Fold*. Default decompositions for these are provided, which create auxiliary lists and iteration constructs to implement the fold and map behavior.

6.3.4 Example

As an illustration, Code listing 6.2 shows part of the generated C++ for a version of Sudoku that uses deferred lists[4]. Without deferred lists, the compiler cannot reason about entire lists at once, and the generated code consists of all loops in unrolled form. Note that as the list with box position offsets is constant, its corresponding loop does get unrolled.

6.4 Evaluation and conclusion

In addition to the high-level modeling language presented in the previous chapter, we added support for deferred parameters, indexable lists and higher-order list processing constructs. Combined with a pseudo-solver that generates C++ code for solving in a later stage, one can now experiment with a problem using the on-line solvers, and switch to the more efficient off-line mode for production use.

Table 6.1 compares the generated code size and corresponding compilation times with those of an earlier version of FD-MCP, which did not yet support

[4]The same model as given in Code listing 5.4, but using a list variable instead of a list of variables, and using the *map* construct instead of list comprehensions

parametrization or higher-order constructs. The benchmarks show that not doing parametrization helps for small instances, but for larger ones it has significant advantages: one does not need to recompile for each instance size, and the generated code size remains constant — both decreasing the required time for compilation. More extensive benchmarks that include comparison of solving times will be given in Section 7.5.

Another observation is that models written in MCP are more concise than in Gecode. This is mainly because less boilerplate code is required, and through the use of higher-order constructs, one can succinctly describe the lists of variables global constraints are posted over. In contrast, in C++ this requires manually setting up a temporary variable, and using an iteration construct to populate it. This can be seen in the code shown in Code listing 6.2.

Code listing 6.2 Generated C++ code for the Sudoku problem using deferred lists

```
using namespace Gecode;

class MCPProgram : public Space {
public:
  IntVarArray cR0;

  MCPProgram() {
    {
      cR0 = IntVarArray(*this,81);
      {
        for (int i=0; i<81; i++) {
          cR0[i].init(*this);
        }
      }
      for (int t0=0; t0<81; t0++) {
        {
          rel(*this,cR0[t0],IRT_GR,0);
          rel(*this,cR0[t0],IRT_LE,10);
        }
      }
      for (int t0=0; t0<3; t0++) {
        {
          for (int t1=0; t1<3; t1++) {
            {
              IntVarArgs cV2_0(9);
              {
                cV2_0[0] = cR0[3*t1+27*t0];
                cV2_0[1] = cR0[1+3*t1+27*t0];
                cV2_0[2] = cR0[2+3*t1+27*t0];
                cV2_0[3] = cR0[9+3*t1+27*t0];
                cV2_0[4] = cR0[10+3*t1+27*t0];
                cV2_0[5] = cR0[11+3*t1+27*t0];
                cV2_0[6] = cR0[18+3*t1+27*t0];
                cV2_0[7] = cR0[19+3*t1+27*t0];
                cV2_0[8] = cR0[20+3*t1+27*t0];
              }
              {
                IntVarArgs as(9);
                for (int asi=0; asi<9; asi++) {
                  as[asi] = cR0[9*(3*t0+t1)+asi];
                }
                distinct(*this,as,ICL_DOM);
              }
              {
                IntVarArgs as(9);
                for (int asi=0; asi<9; asi++) {
                  as[asi] = cR0[3*t0+t1+9*asi];
                }
                distinct(*this,as,ICL_DOM);
              }
              distinct(*this,cV2_0,ICL_DOM);
            }
          }
        }
      }
    }
  }
  MCPProgram(bool share,MCPProgram &s) :
    Space(share,s) {
    cR0.update(*this,share,s.cR0);
  }
};
```

Benchmark		Lines of code				Compilation time (GCC) (s)		
		C++	MCP	MCP (C++)		C++	MCP	
				par.	non-par.		par.	non-par.
allinterval	3	52	22	89	71	3.3	2.7	2.0
	15				179			2.7
queens	4	80	14	81	79	3.4	2.5	2.1
	100				534			>20
partition	4	74	28	105	103	3.3	3.1	2.2
	20				295			4.5
magicsquare	3	62	27	95	92	3.4	2.9	2.8
	6				206			3.1

Table 6.1: Lines of code and compilation times

Chapter 7

Optimizations

The compilation scheme presented in the previous two chapters is responsible for converting (conjunctions of) high-level constraints — represented by Boolean expressions over variables — to low-level, solver-specific variables and constraints.

Often there are many ways to translate a high-level model to a low-level one, with varying efficiency for solving. In fact, the genericity of high-level modeling hides many critical aspects from the user, resulting in poor performance when using a naive approach. Optimizing transformations on the model are performed to overcome this.

One reason for poor naive performance is that primitives of the high-level modeling language do not necessarily coincide with those of the low-level constraint system we are translating for. Sometimes, combinations of constraints in the high-level model — connected using auxiliary variables — can be converted to a single more specific constraint for the low-level system. For example, when a global SUM constraint is provided by the underlying solver, it is typically better to translate $a = b + e \wedge e = c + d$ to a single $a = \text{SUM}([b, c, d])$ constraint, eliminating the e variable, unless it is used in other constraints, needed for branching, or used as a solution.

On the other hand, sometimes it may be necessary to introduce additional variables when the low-level system lacks primitives provided by the high-level language.

The scheme built up so far has some drawbacks in this regard. Among other things:

- It does not handle reified constraints.
- It is unable to perform transformations that span more than one constraint.
- It leaves a lot of common functionality to be implemented by specific *FDSolver* instances.

- One cannot inspect expressions that do not have a corresponding variable in the underlying solver, or branch over them.
- ...

In this chapter, we present several improvements — eventually leading to a general translation framework for constraint models.

7.1 Reification

One unanswered question is reification: the use of Booleans that represent a constraint's truth value — whether or not it holds given a particular assignment for the variables — in other constraints. One case where this occurs is in so-called *disjunctive* models. For example, consider an ad-hoc implementation of the multivalued function "has same absolute value as":

```
eqabs :: Expr Int → FDModel s Int
eqabs x = return x ∨ return (negate x)
```

Using *eqabs*, if one states

```
do
  ax ← eqabs x
  ax @≡ y
```

the resulting *SearchTree* is

```
Branch [ Add (x @≡ y)    true
       , Add (x @≡ −y) true
       ]
```

which is transformed[1] into the expression

```
(x @≡          y) `Or`
(x @≡ negate y)
```

We cannot just use *CLinear* constraints to represent this. What we need is a constraint like *CLinear* which is only conditionally true. The solution is a reified version of *CLinear*:

```
data GConstraint =
  ...
  CLinearR [(IntVar, Int)] GOper Int BoolVar
```

[1] See Section 5.3.2

where

$$[\![CLinearR\ l\ (\oplus)\ c\ b]\!] \equiv \sum_i l_{i,b} * [\![l_{i,a}]\!]\ [\![\oplus]\!]\ c \Leftrightarrow b$$

Here the additional *BoolVar* argument b determines whether the rest of the constraint holds or not.

To support this in the conversion layer, we extend the decomposition system to Boolean variables present in Section 5.6.2. In addition to Code listing 5.11 to compile "top level" Boolean expressions, we have a decomposer for Boolean expressions that are used within other constraints:

```
decompBoolVar :: Expr Bool → FDModel GecodeFD BoolVar
decompBoolVar (Less a b) = do
  reif ← exist
  va ← decompLin a
  vb ← decompLin b
  case (va − vb) of
    Linear l c → addC $ Right $ CLinearR l GLess (−c) reif
  return reif
```

As this is essentially duplicating the code from *convertGecode*, we modify that function to use *decompBoolVar* instead:

```
convertGecode :: Mixin (FDConstraint → FDModel GecodeFD ())
convertGecode self expr = do
  reif ← decompBoolVar expr
  addC $ Right $ CHasValue reif True
```

However, this introduces a new Boolean variable and a reified constraint for each node, regardless of whether it is necessary. For example, it decomposes the simple problem $x \equiv y \wedge y \equiv z$, or $(x\ `Equal`\ y)\ `And`\ (y\ `Equal`\ z)$, into:

$$\begin{aligned} &x \equiv y \Leftrightarrow b_1 \\ &y \equiv z \Leftrightarrow b_2 \\ &b_1 \wedge b_2 \Leftrightarrow b_3 \\ &b_3 \equiv True \end{aligned}$$

Even though it would be sufficient to decompose into the much simpler set of constraints:

$$\begin{aligned} &x \equiv y \\ &y \equiv z \end{aligned}$$

To prevent inefficiencies but avoid code duplication at the same time, we generalize. Instead of both *CLinear* and *CLinearR*, we use a single generalized version:

```
data ReifArg = ConstTrue | Reif BoolVar
data GConstraint =
  ...
  CLinear [(IntVar, Int)] GOper Int ReifArg
```

As *ReifArg* is essentially a wrapper around *BoolVar*, we provide a separate decomposer for *ReifArg* instead of for *BoolVar*. It takes an additional Boolean to signify whether its result is known to be constant:

```
newReif :: Bool → FDModel GecodeFD ReifArg
newReif False = Reif <$> exist
newReif True = return ConstTrue

decompReif :: Expr Bool → Bool → FDModel GecodeFD ReifArg
decompReif (Less a b) isTrue = do
  reif ← newReif isTrue
  va ← decompLin a
  vb ← decompLin b
  case (va − vb) of
    Linear l c → addC $ Right $ CLinear l GLess (−c) reif
  return reif
```

More interesting however, is how *decompReif* deals with Boolean conjunctions:

```
decompReif :: Expr Bool → Bool → FDModel GecodeFD ReifArg
...
decompReif (And a b) True = do
  decompReif a True
  decompReif b True
  return $ ConstTrue
decompReif (And a b) False = do
  vn ← exist
  va ← decompBoolVar a
  vb ← decompBoolVar b
  addC $ Right $ CAnd va vb vn
  return $ Reif vn
```

This essentially propagates knowledge about required validness of the expression being decomposed. In the case of a Boolean conjunction, this results in simply processing the subexpressions as constraints, without introducing an additional variable. To bootstrap this, the actual conversion routine invokes *decompReif*:

```
convertGecode :: Mixin (FDConstraint → FDModel GecodeFD ())
convertGecode self expr = decompReif expr True ≫ return ()
```

7.2 Constraint accumulation

So far, all constraints have always been processed one by one and independently from each other, with only the cache of created low-level variables being shared between different constraints being added. This limits the ability to optimize the model, especially models written by non-experts — which is high-level modeling's main target audience.

One example is models that contain patterns which can be expressed using a global constraint:

$$
\begin{aligned}
&\mathit{demo} = \mathbf{do}\\
&\quad [a, b, c] \leftarrow \mathit{exist}\ 3\\
&\quad a \mathbin{@<} b\\
&\quad b \mathbin{@<} c\\
&\quad c \mathbin{@<} d
\end{aligned}
$$

which is equivalent to SORTED($[a, b, c]$). If each of the $@<$ constraints is translated individually, such a transformation is a priori impossible.

Other, less specific optimizations also require a global view on all constraints; for example:

$$
\begin{aligned}
&\mathit{demo} = \mathbf{do}\\
&\quad [a, b, c] \leftarrow \mathit{exist}\ 3\\
&\quad a + b \mathbin{@\equiv} 10\\
&\quad a * c + b * c \mathbin{@} = 20
\end{aligned}
$$

Use of the distributivity rule can transform the last constraint into $(a+b)*c \mathbin{@\equiv} 20$, which is converted to a *CMult* constraint. However, knowing that $a+b = 10$ would make clear that it is equivalent to $10 * c \mathbin{@\equiv} 20$, allowing a more efficient linear constraint.

The actual optimizations above are introduced in Section 7.3, but first we need to cope with the issue of global translation in general.

The solution is caching the added constraints inside the *FDState*, until they are needed for variables being inspected or the state tested for being failed.

```
data FDState s =
  FDState { ...
          , fdCache ::   FDConstraint
          }
initFDState :: FDState s
initFDState = FDState { ...
                      , fdCache = Const True
                      }
```

This adds a *fdCache* field to the *FDState* ADT (using the record syntax). The rest of the *Solver* instance for *FDinstance s* can now be given:

```
instance FDSolver s ⇒ Solver (FDWrapper s) where
  type Constraint (FDWrapper s) = Either FDConstraint (Constraint s)
  type Label (FDWrapper s)      = FDLabel s
  run (FDWrapper s) =  run $ evalState s initFDState
  add (Left m)      = do state ← get
                         put state {fdCache = fdCache state @@& m}
  add (Right c)     = wrap (add c)
  isFailed          = commit ≫ checkFailed
  mark              = FDWrapper $ do
                        iLabel ← lift mark
                        oLabel ← get
                        return $ FDLabel oLabel iLabel
  goto (FDLabel o i) = FDWrapper $ do
                        put o
                        lift $ goto i
```

commit here is a function that processes the expression stored in *fdCache state*, "flushes" it to the underlying solver, and resets *fdCache* to *Const True*:

```
commit :: FDWrapper s ()
commit = do
  let toProc = fdCache state
  modify (λstate → state {fdCache = Const True})
  compile_constraint toProc
```

The alternative is performing an extra *tree2model* on the entire initial search tree before solving. This indeed merges all constraints into a single expression, but it is not compatible with search, as *tree2model* does not support dynamic nodes. By doing the accumulation inside the FD wrapper, it is transparently available to every solver.

7.3 Graph-based transformation and conversion

The next step is the implementation of *compile_constraint*, which receives a conjunction of all accumulated constraints. The input is represented by an expression tree. This is however again not the most optimal format for the transformations we envision. To perform optimizations and other transformations, we require convenient access from a constraint instance to its variables and vice versa. Therefore we break down the expression tree into a set of individual constraints, and combine these into a graph which makes the connections between constraints explicit: the constraint network graph (CNF).

In a second stage, the pieces are recombined (possibly in a different way) to constraints for the low-level system, by pattern matching on parts of this intermediary graph.

7.3.1 Constraint network graph

As explained, the model is transformed to a constraint network graph. All expressions and subexpressions of the original model become separate variables, regardless of how they were introduced.

First, all subexpressions are replaced by additional variables and constraints, comparable to the complete decomposition as introduced in Section 5.6.1. For example, $a < b + c$ would be broken up into the conjunction $a < t_1 \wedge t_1 = b + c$.

Then these constraints are combined into a graph. Each variable (original or freshly introduced) is mapped to a graph node, and each constraint to an edge — labeled with its constraint type. Since constraints are not necessarily binary, in general the edges are hyperedges and the graph a hypergraph.

Higher-order expressions To encode higher-order expressions like the iteration constructs from Section 6.3.3, we allow certain (hyper)edges to be labeled with a subgraph that represents the constraint network graph of the inner expression. This has several advantages over replacing them by the sequence of direct constraints they represent (also called flattening or more specifically loop unrolling):

- It allows the additional graph-based optimizations to be performed on the generic problem, before instantiation of parameters.
- Creating a graph of the flattened model may well be very expensive, since its size is proportional to the number of variables and constraints in the instance.
- Unrolling is not possible if loop or array sizes depend on a parameter that is not known at Haskell-runtime.

These internal graphs can have some nodes marked as "imported" from the parent model. For example, the model $b = \mathit{sum}\ a$ evaluates to the expression $b = \mathit{fold}\ (a, 0, +)$, and the model $b = \mathit{map}\ (\lambda x \to x + c)\ a$ evaluates to $b = \mathit{map}\ (a, +c)$. The corresponding constraint network graphs are shown in Figure 7.1. The boxes and arrows represent the hyperedges/constraints, while the circles represent the nodes/variables. The boxes for *fold* and *map* are labeled with a new graph, with some nodes marked specially (*init*, *arg* and *result*).[2] Since the variable c is not local to the *map*'s function argument (like x), it is imported from the parent model's N_c as N'_c.

[2] A similar technique is used for quantified variables in e.g., *forall*.

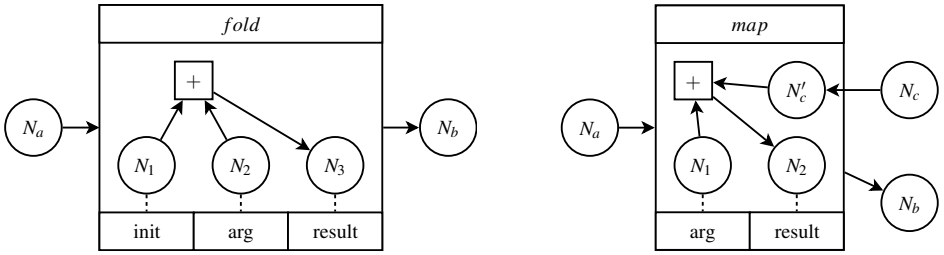

Figure 7.1: submodels: $b = fold(a, 0, +)$ and $b = map(a, +c)$

Translation from an expression tree to such a graph proceeds as follows:

- For every node in the tree a corresponding node in the graph is created. To preserve variable identities, variables in the tree are reused as their corresponding nodes in the graph.
- The relation between a tree node and its children is materialized as an edge between the corresponding graph nodes.

For example, consider the expression $a + b$. The corresponding graph contains three nodes: N_{a+b}, N_a and N_b, with a hyperedge labeled + among them.

Note that multiple occurrences of the same variable in the tree end up sharing the same node in the graph, abandoning the tree invariant. The possibility of sharing is further exploited by mapping identical subtrees onto the same graph node. This is implemented by keeping a map from (sub)expressions to graph nodes in a state threaded through the conversion process, comparable to the *intCache* and *boolCache* construct from Section 5.4.4. If a subtree is encountered that is identical to an already-processed subtree, its graph node is reused and only an edge is added to it. This reusage of graph nodes is at least required for leaf nodes corresponding to constraint variables, since otherwise information about constraint variables occurring in more than one constraint would be lost. Extending the mechanism to other nodes essentially results in CSE (common subexpression elimination), which is more thoroughly explored in (Rendl 2010).

Note that in the model, constraints are represented by Boolean expressions. Their return values denote their truth value. This means that they are all explicitly reified in the constraint network form. Since the point of a CSP is finding an assignment for the variables which causes the model expression to evaluate to true, the truth value for the root node incurs an additional constraint, equating it to the constant "true".

Constant propagation As explained in Section 7.1, we do not want everything to be translated to reified constraints. These are often less efficient, or may

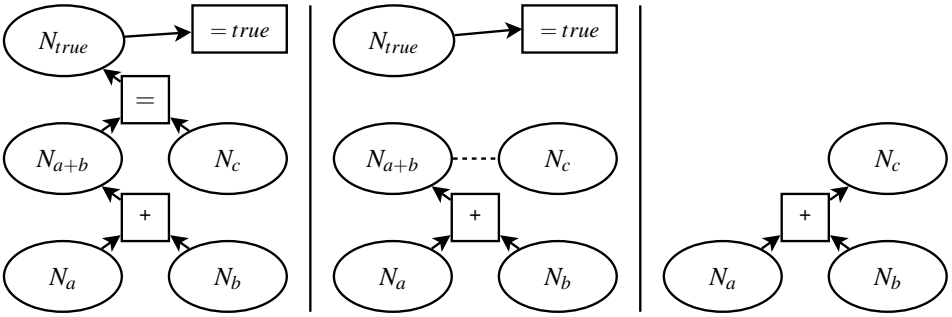

Figure 7.2: Optimization: unification of equal nodes

not always be supported by solvers. However, by exploiting the richer structure the graph representation has, we can improve the mechanism from the previous section: instead of passing known truth values along during the decomposition process, we can do a full *constant propagation* step afterwards.

When a model consists of $c_1(a, b) \wedge c_2(a, b)$, a naive translation creates a constraint network with three nodes for Boolean variables in addition to a and b: v_{12}, v_1 and v_2. The generated constraints are $v_{12} = true$, $v_{12} = v_1 \wedge v_2$, $c_{1,reif}(a, b, v_1)$ and $c_{2,reif}(a, b, v_2)$. Since v_{12} has value $true$, and $v_{12} = v_1 \wedge v_2$, v_1 and v_2 themselves must have the value $true$. Comparable reasoning can be used to infer values when *or* and *not* constraints are present. Although not strictly necessary, this information helps choosing non-reified constraints when not required.

Unification Constant propagation further enables a simplification that compensates for the implicit reification that is inherent to expressing all constraints as Boolean expressions. When an equality constraint between two nodes exists, with a truth value that can be proving to be true using constant propagation, the constraint can be dropped and the nodes unified. An example will clarify the matter: the constraint model $c = a + b$ over the three variables a, b and c would be turned into a graph with 5 nodes (N_a, N_b, N_{a+b}, N_c and N_{true}), and 3 constraints: $N_a + N_b = N_{a+b}$, $(N_{a+b} = N_c) = N_{true}$ and $N_{true} = true$. In this case, the N_{a+b} and N_c nodes can be unified, which results in the simpler model with four nodes (N_a, N_b, N_c, N_{true}) and two constraints ($N_a + N_b = N_c$) and $N_{true} = true$. Clearly the fourth node and the second constraint are redundant, simplifying the result to a single edge over three variables: $N_a + N_b = N_c$.

7.3.2 Constraint tiling

In the first phase of the compilation process, the high-level solver-independent model has been disassembled into a solver-independent constraint network graph for easier optimization. Now, in the last phase, the constraint graph is turned back into a constraint model, but now of a solver-dependent and lower-level nature. The nodes and edges of the graph are mapped to constraint variables and constraints of the underlying constraint solver.

Our conversion algorithm is a *tiling* process, akin to instruction tiling in compiler back-ends (Appel 1998), that matches subgraphs against "tiles" supported by the underlying solver. While we assume that small tiles are available for every type of node and edge in the graph, preference is given to larger tiles that cover multiple nodes and edges. Note that the main difference with traditional tiling is that we consider a graph structure rather than a tree structure. While this may somewhat complicate matters, there is potentially more information available to make good tiling decisions.

In a first step, we decide which nodes in the graph become the variables of the resulting constraint problem, and how other nodes can be written as a function of those, by "absorbing" edges into annotations on nodes. In a second step, these annotations are used together with the remaining edges to generate the final low-level constraints.

Example Assume the following model expression:

$$x + y + z \leqslant z - y \tag{7.1}$$

Further assume the underlying solver supports linear inequalities:

$$\sum_{i=1}^{n} a_i v_i \leqslant c \tag{7.2}$$

where a_i and c are constants, and v_i are variables. Clearly, the given constraint (7.1) can be translated to a single linear inequality constraint, with $\bar{a} = [1, 2, 0], \bar{v} = [x, y, z], c = 0$, or even simpler, with $\bar{a} = [1, 2], \bar{v} = [x, y], c = 0$.

The calculated constraint network (see Figure 7.3) has an edge corresponding to the inequality, with nodes corresponding to $x + y + z$ (N_2) and $z - y$ (N_3). Furthermore, N_2 is connected using a $+$ edge to the nodes corresponding to $x + y$ (N_1) and z. To be able to match this as a single linear inequality, it is necessary to know that all these nodes can be considered linear combinations of other nodes. We capture this information in a node annotation $linear(\bar{a}, \bar{v}, c)$, meaning:

$$linear(\bar{a}, \bar{v}, c) = c + \sum_{i=1}^{n} a_i v_i \tag{7.3}$$

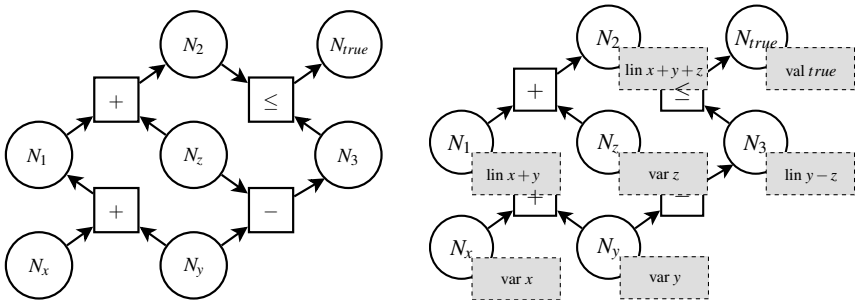

Figure 7.3: Constraint Network Graph for $x + y + z \leqslant z - y$, unannotated and annotated

Recognizing that N_2 is connected using a *plus* edge to two other nodes, one can try to find out whether those two nodes can be considered linear combinations of other nodes as well (or simple constants or variables). Since one of them is again connected using a *plus* to N_1, this matching continues recursively while traversing the graph. Eventually, nodes are reached that can no longer be considered linear combinations of other (not yet explored) nodes. These end nodes become constraint variables, while all nodes on the paths from the inequality to the end nodes, are considered linear functions of others. All this information is materialized as an annotation on graph nodes, to avoid later recomputation. Finally, using the following formula, the inequality is turned into the obvious single linear inequality constraint:

$$linear(\bar{a}_1, \bar{v}_1, c_1) \leqslant linear(\bar{a}_2, \bar{v}_2, c_2) \Leftrightarrow \sum_{i=1}^{n} a_{1i} v_{1i} + \sum_{i=1}^{m} (-a_{2i}) v_{2i} \leqslant c_2 - c_1 \quad (7.4)$$

There are many more useful examples of annotations, including (potentially parametrized) constant values, known sizes of array variables, and certain structures imposed by global constraints (such as *alldifferent*).

To recognize the patterns, we build them all simultaneously. We create an annotation for every node in the graph, potentially using a trivial annotation that describes it as a simple variable. To do so, we put all potential annotations (called annotation generators) in a priority queue and process them one by one, starting from those that have a chance to create a "better" annotation (one for a constant value annotation is given highest priority, and one for a separate variable lowest priority). Annotation generators may depend on the presence of specific annotations on other nodes, in which case the corresponding generator may be called prematurely, and removed from the queue.

There is a further complication: not each type of annotation may be implemented or even be useful for every edge. For example, there is typically no

constraint that implements a division between linear combinations of variables. Therefore the *division* edge should not support the linear annotation. Which annotations are possible and useful depends on the solver.

The resulting algorithm is as follows:

1. For each edge type, the solver interface is asked which annotation types it supports for its vertexes, and which annotation generators are available.

2. Using a partial ordering on these generators, they are processed one by one:

   - The generator is skipped if it would not produce any useful annotation. This is the case if it is for a node which is already fully specified (annotations compatible with all its edges have been created already).
   - The generator can request annotations (of a particular type) of neighboring nodes. This fails if a dependency loop occurs or no generator for that node/type combination exists. Otherwise the respective generator is invoked, its resulting annotation stored, and passed back to the calling generator.
   - Based on this information, the calling generator as a whole may fail, or create the annotation.
   - If this results in a second annotation for a same node, an artificial equality constraint between it and the earlier one is added.

3. New constraint variables are created for all remaining nodes.

4. Finally, the remaining edges and annotations for their vertexes are passed to the solver interface to produce low-level constraints.

The result is an eager matching algorithm that consumes edges in the graph by describing some nodes in it in function of other nodes, possibly recursively. Nodes for which this is not possible become simple constraint variables, and edges for which this is not possible become real constraints.

Another advantage of this scheme is modularity: instead of requiring a complete *compile_constraint* implementation for each solver back-end, it is now global, and only specific details are requested from the instance (annotation generators and translation of individual unabsorbed edges). The entire decomposition logic, as well as several optimizations, is invisible to the solver-specific code.

7.3.3 Complexity analysis

We analyze the complexity of the constraint tiling algorithm. The entire process consists of several steps:

- For each edge, the solver interface is asked for annotation generators. We assume this results in a constant time pattern match on the edge type. $\mathcal{O}(e)$, with e the number of edges.
- All annotation generators are sorted, based on their priority. $\mathcal{O}(g \log g)$, with g the number of generators..
- Additionally, an index is maintained that maps (annotation type, node)-pairs to generators. Assuming a tree-based map, $\mathcal{O}(g \log g)$.
- During the main loop, each annotation generator fires at most once. We assume the actual execution of a generator — apart from invocation of other generators — is dominated by one recursive lookup for another generator per connected variable (vertex). $\mathcal{O}(g\,v \log g)$, with v the maximum number of vertexes per edge.
- Without the above optimization steps, more edges would remain in the constraint graph, and each has to be converted to a constraint anyway.

In all, the extra cost incurred by performing the tiling algorithm is

$$\mathcal{O}(A\,e + B\,g \log g + C\,g\,v \log g)$$

Assuming most edges result in at least one generator, this is dominated by $\mathcal{O}(g\,v \log g)$. If the number of generators per edge is limited to a constant, this simplifies to $\mathcal{O}(v\,e \log e)$.

If higher-order expressions are used, the constraint tiling algorithm is run for each subgraph separately, with knowledge from the parent graph(s) represented in the subgraph as special constraints (edges) on the imported vertexes. Therefore, the complexity of all constraint network graphs needs to be summed together.

This low complexity is only possible because the matching algorithm uses a heuristic-based execution order of the different generators. The alternative, using backtracking search to find the best order, must in a worst-case situation try $\mathcal{O}(g!)$ permutations, which is worse than exponential in the number of edges. In reality, the order has a limited impact on the produced low-level model, and experiments have shown that a heuristics-based approach suffices.

7.3.4 Example

To demonstrate the effectiveness of the translation, consider the *AllIntervals* problem shown in Code listing 7.1. The aim is to find a sequence of numbers of size n, where each number is different, between 0 and $n-1$, and the absolute values of differences between subsequent elements take all values between 1 and $n-1$. Using monadic composition, we are able to abstract the creation of the difference

Code listing 7.1 The AllIntervals example

```
  -- diff: the differences between successive elements of an array
diff l = do
  d ← exists                          -- request an (array) variable d
  let n = size l                      -- introduce n as alias for size l
  size d @= n − 1                     -- size of d must be one less than n
  loopall (0, n − 2) $ λi → do        -- for each i in [0..n-2]
    d ! i @= abs (l ! i − l ! (i + 1)) -- d[i] = abs(l[i]-l[i+1])
  return d                            -- and return d to the caller
allint :: FDModel s (Expr [Int])      -- type signature
allint n = do                         -- function 'model' takes argument n
  x ← exists                          -- request an (array) variable x
  size x @= n                         -- whose size must be n
  d ← diff x                          -- d is the "diff" of x
  x `allin` (0, n − 1)                -- all x elements are in [0..n-1]
  d `allin` (1, n − 1)                -- all d elements are in [1..n-1]
  allDiff x                           -- all x elements are different
  allDiff d                           -- all d elements are different
  x @!! 0 @< x @!! 1                  -- some symmetry breaking
  d @!! 0 @> d ! (n − 2)              -- some symmetry breaking
  return x                            -- return the array itself
```

array in a separate function (*diff*), even though it introduces an additional variable. Also note the use of *size l* within the *diff* function. The implementation only supports arrays for which a (parametrized) constant size can be derived. Because of node unification in the constraint graph, this is easy: it is equal to n, enforced by *size x* @= n in *allint*.

7.4 Related work

Different languages and systems exist that perform automated translation from high-level CP models to solver-specific programs:

- Like FD-MCP, the Tailor (Rendl 2010) system attempts to present a high-level modeling interface to the user, with optimizations to prevent inefficient solving caused by users unfamiliar with the intricacies of CP. However, it is a separate tool that processes specific modeling languages (Essense' and XCSP), and translates them to Minion, FlatZinc or Gecode programs. It

lacks the flexibility MCP provides when solving directly, yet provides rather advanced model optimizations.

- The Zinc (Marriott et al. 2008) family of languages (including MiniZinc and FlatZinc) provide an extensive framework for translating from high-level models to flattened and solver-specific input. The translation (Stuckey, De La Banda, Maher, Slaney, Somogyi, Wallace, and Walsh 2005) is based on a rule-based system called ACD Term Rewriting (Duck, Stuckey, and Brand 2006), which performs transformations directly on the syntax tree. Zinc provides comparable higher-order constructs (only over arrays of known length), and converts Boolean combinations of constraints to reified constraints.

- The multi-paradigm Mozart (Mozart Team 2004) programming system, based on the Oz language, includes support for constraint programming in a flexible and declarative way. Constraint solving concepts such as spaces are provided first-class in the system. It seems however mostly focused on the ability to program search and propagation, and lacks high-level modeling features such as writing constraints as expressions.

- The GELATO (Cipriano, Gaspero, and Dovier 2009) system allows models to be solved through a hybrid approach consisting of both CP and LS (Local Search). Models that can be specified using Prolog or MiniZinc, or solving using a combination of Gecode (CP) and EasyLocal++ (LS).

7.5 Evaluation

The FD-MCP system contains an implementation[3] of the optimizations described in this chapter. In particular, several annotation types are present, including:

- known constant (potentially parametrized) value: $v_1 == f(p_a, p_b, \ldots)$
- *conditionally* known constant value: $f(p_a, p_b, \ldots) \Rightarrow v_1 == g(p_c, p_d, \ldots)$. When v_1 is used as condition for a reified constraint further on, it is translated to code within an `if` block in C++
- known size of a list variable
- linear combinations of integer variables
- separate Boolean and integer variables

Additionally, for Gecode-based solvers (either on-line or off-line), specific optimizations are applied by pattern-matching on (parts of) the subgraph of certain

[3] See http://users.ugent.be/~tschrijv/MCP/

higher-order edges. For example, a *Fold* that reduces a list using the (+) operator is translated to a single efficient sum constraint propagator.

As this chapter only concerned optimizations in the intermediate layer and back-ends, the front-end language presented in the previous two chapters remains valid.

Benchmarks To verify the quality of the translation, a set of classic CP problems were ported from C++ to FD-MCP, resulting in smaller code on average, and benchmarked. We compare the runtimes of original C++ Gecode implementations against runtimes of our Gecode-backed solvers, as well as the runtime of the code generated by the C++ code generation pseudo-solver.

Table 7.1 and Table 7.2 show the measured solving times[5]. Time-outs (more than 120s of runtime) are listed as "-". Table 7.3 shows code generation time, compilation times and lines of code (not including instance data, output routines or main functions). The columns marked a) refer to original Gecode benchmarks in C++, those marked b) to FD-MCP-generated C++, those marked c) to direct solving using FD-MCP's fixed search, and those marked d) to direct solving using FD-MCP's programmed search. The numbers clearly show that generated C++ code has very close or even slightly better performance than the original benchmark. The latter does some additional bookkeeping and has more options, sometimes resulting in slightly higher overhead. On the other hand, the generated code sometimes contains a small amount of superfluous variables, causing inefficiencies. Before higher-order constructs were introduced, this amount was much higher, with significantly slower solving as a result. When comparing with the direct solvers, larger differences occur. When using Gecode's search, the overhead of switching from Haskell to C++ and back for each constraint can become significant, eg. in the *queens* benchmark. When using MCP's search, this overhead also occurs during search, resulting in significantly lower performance.

[4]csplib 7: allinterval, csplib 28: bibd (6,10,5,3,2), efpa: (5,3,2,4), csplib 6: golomb rulers, csplib 19: magic square and magic sequence, csplib 49: partition

[5]Benchmarks performed on a Intel Core 2 Duo E8500 system with 4GiB RAM, running 64-bit Ubuntu 10.04, GHC 6.12.1, Gecode 3.2.1 and MCP 0.7.0. C++ benchmarks were modified to use the same search order and strategy as FD-MCP. Runtimes are averages over running each instance for 10 minutes.

name[4]	size	Solving time (s)			
		C++	MCP		
		(a)	C++(b)	Search (c)	Run (d)
allinterval	7	0.0041	0.0041	0.0087	0.011
	8	0.0045	0.0047	0.0099	0.016
	9	0.0066	0.0069	0.013	0.035
	10	0.016	0.018	0.025	0.12
	11	0.065	0.077	0.084	0.56
	12	0.32	0.38	0.38	2.8
	13	1.8	2.1	2.1	15
	14	10	12	12	84
	15	61	71	72	-
alpha			0.0045	0.022	0.026
bibd		0.0041	0.0053	0.11	0.11
domino		0.0096	0.0098	0.064	0.09
		0.044	0.041	0.097	0.52
efpa		0.0045	0.0051	0.09	0.093
golombruler	6	0.0042	0.0042	0.027	0.031
	7	0.0074	0.0083	0.04	0.067
	8	0.042	0.048	0.089	0.27
	9	0.35	0.4	0.44	1.7
	10	3.2	3.5	3.4	13
	11	81	90	82	-
graphcolor		0.12	0.12	1.2	1.2
grocery		0.099	0.099	0.094	0.099
langford	10 3	0.009	0.009	0.04	0.045
magicseries	10	0.004	0.004	0.0099	0.01
	20	0.0041	0.0041	0.016	0.017
	50	0.0046	0.0044	0.036	0.039
	75	0.0055	0.0049	0.053	0.059
	100	0.0067	0.0054	0.073	0.082
	200	0.017	0.0096	0.16	0.19
	375	0.061	0.027	0.34	0.43
	500	0.12	0.049	0.52	0.67
	1000	0.94	0.29	1.7	2.2
magicsquare	3	0.0039	0.0039	0.012	0.012
	4	0.0082	0.0092	0.021	0.079
	5	0.4	0.48	0.62	110
	6	0.0043	0.0043	0.022	-
	7	2.6	2.9	4.5	-
minesweeper		0.004	0.0044	0.43	0.44

Table 7.1: Benchmark results: solving time for C++ benchmarks, and for corresponding FD-MCP programs (off-line mode, and both on-line modes)

name	size	Solving time (s)			
		C++	MCP		
		(a)	C++(b)	Search (c)	Run (d)
partition	10	0.0046	0.0044	0.011	0.023
	14	0.0088	0.008	0.018	0.39
	18	0.019	0.016	0.036	9.8
	20	0.054	0.046	0.086	33
	22	0.16	0.14	0.24	160
	24	0.55	0.47	0.8	-
	26	0.57	0.48	0.83	-
	28	1.9	1.6	2.8	-
	30	5	4.3	7.3	-
	32	6.5	5.6	9.6	-
queens	8	0.004	0.0039	0.019	0.02
	10	0.004	0.0039	0.026	0.028
	13	0.0043	0.0042	0.042	0.043
	21	0.0041	0.0041	0.1	0.16
	34	0.0046	0.0044	0.25	-
	55	0.008	0.0076	0.65	-
	70	0.0082	0.0074	1	-
	89	0.012	0.011	1.7	-
	100	0.015	0.013	2.1	-
	111	0.089	0.089	2.8	-

Table 7.2: Benchmark results: solving time for C++ benchmarks, and for corresponding FD-MCP programs (off-line mode, and both on-line modes) (continued)

name	Compile/codegen time (s)				Lines of code		
	codegen	GCC		GHC	C++		Haskell
	(b)	(a)	(b)	(b-d)	(a)	(b)	(b-d)
allinterval	0.0063	1.3	0.87	0.032	26	113	18
alpha	0.023		1.1	.045		276	30
bibd	0.01	1.3	0.88	0.042	87	146	16
domino	0.022	1.4	0.99	0.041	61	140	28
efpa	0.0099	1.8	0.96	0.044	105	177	15
golombruler	0.01	1.3	0.9	0.044	41	149	23
graphcolor	0.022	1.3	0.86	0.032	55	124	15
grocery	0.0063	1.3	0.86	0.039	26	96	8
langford	0.0078	1.3	0.88	0.033	77	121	19
magicseries	0.0063	1.3	0.87	0.033	28	120	11
magicsquare	0.011	1.4	0.89	0.043	37	116	20
minesweeper	0.012	1.3	0.86	0.033	78	113	20
partition	0.0062	1.4	0.91	0.043	43	138	24
queens	0.0067	1.3	0.85	0.033	26	104	10

Table 7.3: Benchmark results: compilation time and lines of code

Chapter 8

Language design

In this chapter, we compare the design of the front-end language defined in Chapters 5 and 6 with that of another modeling language, MiniZinc.

8.1 Example

Code listing 8.1 and Code listing 8.2 show a full program for solving the earlier N queens problem (see Code listing 6.1), in both MiniZinc and FD-MCP.

MiniZinc The MiniZinc example first declares a global parameter (`n`) and array of constraint variables (`q`). Then it defines a predicate `noattack`, and enforces it for all numbers `i` and `j`. Finally, it asks to label the variables in q, and report satisfying solutions.

FD-MCP The FD-MCP approach is very similar, except the source code describes a program, and not just a constraint problem. One of the consequences is the lack of global declarations: instead of the size parameter `n`, we have a model function argument n, and instead of a global array of constraint variables, these are part of the model description function, and returned.

Although both systems support influencing the labeling and search, these are not shown in the above example, as search will be more thoroughly explored in the next chapters. Instead, default labeling and search are used — *example_sat_main_single_expr* is a helper function that solves a satisfaction problem which returns an list of constraint variables, and takes a single integer parameter as argument.

Code listing 8.1 Full N queens problem in MiniZinc

```
int: n;

array [1..n] of var 1..n: q :: is_output;

predicate
    noattack(int: i, int: j, var int: qi, var int: qj) =
        qi     != qj     /\
        qi + i != qj + j /\
        qi - i != qj - j;

constraint
    forall (i in 1..n, j in i+1..n) (
        noattack(i, j, q[i], q[j])
    );

solve :: int_search(q) satisfy;
```

Code listing 8.2 Full N queens problem in FD-MCP

```
import Control.CP.FD.Example
noattack i j qi qj = do
  qi     @≠ qj
  qi + i @≠ qj + j
  qi - i @≠ qj - j
nqueens :: ExampleModel ModelInt
nqueens n = do
  q ← exists
  size p @ = n
  q `allin` (0, n − 1)
  allDiff q
  loopall (0, n − 2) $ λi →
    loopall (i + 1, n − 1) $ λj →
      noattack i j (q ! i) (q ! j)
  return p
main = example_sat_main_single_expr nqueens
```

8.2 Feature comparison

In what follows, we do a side-by-side comparison of features of both systems.

Constraint variables and constraints Both systems support several types of constraint variables: Booleans and integers, and arrays/lists of them. MiniZinc additionally supports set variables. The number of constraint types supported is significantly larger in MiniZinc, through the use of a library of global variables, together with default decompositions. Adding new constraints to FD-MCP is easy, though. An additional constraint was already implemented in the context of a bachelor project.

Problem classes and instances The presence of parameters allows problem descriptions to be written only once for an entire class of problems, only requiring values for some parameters to instantiate a problem instance. MiniZinc allows global parameters (like `n` above) to defined, whose value is specified in a separate (instance) file. However, it only operates on fully-instantiated problems, requiring a separate run of the translation routines for each instance.

As problems are specified in FD-MCP using first-class model trees, parameters can trivially be abstracted from model definitions by writing them as functions that take the parameter as argument. Using deferred lists and explicit higher-order constructs (see below), FD-MCP can reason about not fully instantiated problems — doing the translation once for an entire problem class.

Higher-order constructs To model repeating structures in the problem, both systems have support for higher-order constructs. MiniZinc, among others, has constructs for conjunctions (`forall (...)`) and summing (`sum (...)`). Special syntax helps constructing arguments to these operations; for example the `(i in 1..n, j in i+1..n) noattack(i, j, q[i], q[j])` construct above. This is syntactic sugar for a more generic construct: array comprehensions. The example above could also have been written using a comprehension as `[noattack(i,j, q[i], q[j]) | i in 1..n, j in (i+1)..n]`.

To compare with FD-MCP, we need to make the distinction between compilable, parametrized models, and others. In the former case, we are limited to constructs that can be represented explicitly in FD-MCP's model tree (*loopall*, *fold* and *map*), and all constructs that can be derived from these core operations (e.g. *sum*, *count*). However, in the normal mode of operation, where lists are not deferred and fully known at run-time, all Haskell's higher-order and list operations are implicitly available, which also includes list comprehensions that are very similar to those available in MiniZinc. This is possible because all expressions and sub-models in FD-MCP models are first-class Haskell values, that can be passed to other functions.

8.3 Beyond satisfaction problems

MiniZinc expects a source file to represent a single constraint problem (or class), while FD-MCP uses Haskell source code that represents potentially complex full programs. This allows for more concise descriptions in MiniZinc, as its language only requires domain-specific syntax and nothing more. However, having a full language at one's disposal permits more complex interaction with the problem and its solutions.

All examples shown so far in this thesis concerned simply searching for solutions to satisfaction problems. Although useful for explaining the actual constraint modeling system, it hides this advantage of using a full language.

Generating Sudoku puzzles Contrary to Code listing 5.4, where we were looking for solutions to a Sudoku puzzle, we may want to *generate* such puzzles. Ignoring constraints such as a desired (human) difficulty level, generating a puzzle is conceptually very easy:

1. find a random solution to an empty puzzle (one without fields filled in)

2. remove fields from it as long as no more than one solution is possible

To keep the type signatures short, we fix the type of the constraint solver being used:

```
type Sol = Gecode
```

Then, we need a generic model for Sudoku itself:

```
sudoku :: [(Int, Int)] → SearchTree Sol [Expr Int]
sudoku fields =
  do mat ← exist 81
     mat `allin` (1, 9)
     let row i = [mat !! (i * 9 + p) | p ← [0 .. 8]]
         col i = [mat !! (i + 9 * p) | p ← [0 .. 8]]
         blkPos = [0, 1, 2, 9, 10, 11, 18, 19, 20]
         blk r c = [mat !! (3 * c + 27 * r + p) | p ← blkPos]
     forM_ [0 .. 2] $ λi →
       forM_ [0 .. 2] $ λj → do
         allDiff $ row $ i + 3 * j
         allDiff $ col $ i + 3 * j
         allDiff $ blk i j
     forM_ fields (λ(pos, val) → mat !! pos @= val)
     return mat
```

This code is almost identical to the one given in Code listing 5.4, except that it takes an additional $[(Int, Int)]$ argument that represents the already assigned fields.

To retrieve the list of assignments for an empty Sudoku problem, we use the *label* function from Section 4.1.6:

```
sudokuFields :: SearchTree Sol [(Int, Int)]
sudokuFields = do
  mat ← sudoku []
  vals ← label mat
  return $ zip (0..) vals
```

where *zip* is the function that combines elements from two lists into a list of tuples. For example *zip* $[1, 2, 3]$ $[5, 6, 7]$ equals $[(1, 5), (2, 6), (3, 7)]$. This turns a solution into a list that can be passed as argument to *sudoku* again.

Then we define a helper function to check whether a search tree corresponds to a solved state for its returned list of variables:

```
isSolved :: SearchTree Sol [Expr Int] → Bool
isSolved model = case (dfsSearch model) of
  []  → False   -- no solutions
  [x] → True    -- single solution
  _   → False   -- more solutions
```

Note that we could have used *length* (*dfsSearch model*) here, but that would search for all solutions, while we only need to know whether there are zero, one or more. Next, we define a function to compute all sublists of a given list with a single element removed:

```
sublists :: [a] → [[a]]
sublists []     = []
sublists [x]    = [[]]
sublists (x : xs) = xs : (map (x:) $ sublists xs)
```

For example, *sublists* $[1, 2, 3]$ is equal to $[[1, 2], [1, 3], [2, 3]]$.

Using these, we can write a modified Sudoku model which, starting from a list of assignments that result in a single solution, searches for a minimal sublist that results in the same solution:

```
sudokuPrune :: [(Int, Int)] → SearchTree Sol [(Int, Int)]
sudokuPrune fields = do
  let toPrune = filter (isSolved ∘ sudoku) $ sublists fields
  case toPrune of
    [] → return fields
    _ → disj $ map sudokuPrune toPrune
```

which works by first computing the sublists that still have a single solution, and branching over those if possible. If this is no longer possible, it is considered a solution.

Finally, *sudokuFields* and *sudokuPrune* can be bound together to do everything in one step:

$$main = print \$ head \$ dfsSolve \$ sudokuFields \gg\!= sudokuPrune$$

which prints the first found minimal assignment. This will work, but using default search and labeling orderings, the solutions will not look very random. Possible improvements include shuffling the assignments before passing them to *sudokuPrune*, or using a random search order.

8.4 Conclusion

We've shown how a similar degree of abstraction can be achieved when writing a constraint model in FD-MCP, compared to MiniZinc. FD-MCP is less feature-rich in terms of supported constraints and variable types. However, FD-MCP's internal representation supports uninstantiated problems, which allows it to perform problem-class-wide optimizations when generating off-line code. Finally, we have shown how the ability to having constraint modeling code and Haskell interact directly allows for very elegant modeling of Sudoku problem generation — something that is not possible in a non-embedded DSL like MiniZinc.

Part III

Search Modeling

Chapter 9

Search combinators

So far, we have mainly talked about the high-level modeling of the constraint problem itself. In what follows, we deal with the other part of CP, namely search.

9.1 Rationale

Search heuristics often make all the difference between effectively solving a combinatorial problem and utter failure. Heuristics make a search algorithm efficient for a variety of reasons, e.g., incorporation of domain knowledge, or randomization to avoid heavily tailed runtimes. Hence, the ability to swiftly design search heuristics that are tailored towards a problem domain is essential for performance.

Contrary to modeling of the combinatorial problem itself (See Part II), for which a range of high-level modeling languages exist, little work exists around the formulation of accompanying search heuristics. Either the design of search is restricted to a small set of predefined heuristics (e.g., MiniZinc (Nethercote, Stuckey, Becket, Brand, Duck, and Tack 2007)), or it is based on a low-level general-purpose programming language (e.g., Comet (Van Hentenryck and Michel 2005)). The former is clearly too confining, while the latter leaves much to be desired in terms of productivity, since implementing a search heuristic quickly becomes a non-negligible effort. This also explains why the set of available heuristics is typically small: it takes a lot of time for CP system developers to implement heuristics, too – time they would much rather spend otherwise improving their system.

9.2 Introduction

The *Search combinators* approach resolves this stand-off between solver developers and users, by defining a high-level search language, and easy to implement

semantics for it.

A compositional approach allows for expressing complex search heuristics based on an (extensible) set of primitive combinators. At the same time, the modular design makes it easy to implement.

Several proposals have already been made for high-level search specification languages (e.g., (Samulowitz, Tack, Fischer, Wallace, and Stuckey 2010)). The Search combinators approach differs in that it bridges the gap between a conceptually simple language (high-level, functional and naturally compositional) and an efficient implementation (low-level, imperative and highly non-modular).

The primitives of the search language are implemented as *mixin* components. As in Aspect-Oriented Programming, mixin components neatly encapsulate the *cross-cutting behavior* of primitive search concepts, which are highly entangled in conventional approaches. Cross-cutting means that a mixin component can interfere with the behavior of its sub-components (in this case, sub-searches). The combination of encapsulation *and* cross-cutting behavior is essential for systematic reuse of search combinators. Without this degree of modularity, minor modifications require rewriting from scratch.

An added advantage of mixin components is extensibility. New features can be added to the language by adding more mixin components. The cost of adding such a new component is small, because it does not require changes to the existing ones. Finally, this approach is solver-independent and therefore makes search combinators a potential standard for designing search. The code is available at `http://users.ugent.be/~tschrijv/SearchCombinators/`.

9.3 High-Level Search Language

This section shows the high-level search language and its expressive power. The concrete syntax used here consists of nested terms, compatible with the *annotation* language of MiniZinc (Nethercote, Stuckey, Becket, Brand, Duck, and Tack 2007).

The *expression language* comprises the typical arithmetic and comparison operators and literals that require no further explanation. Note however that it allows references to the constraint variables and parameters of the underlying model.

9.3.1 Primitive Search Heuristics

The search language is used to define a *search heuristic*, which a *search engine* applies to each node of the search tree. For each node, the heuristic determines whether to continue search by creating child nodes, or to prune the tree at that node.

The search language features a number of primitives, listed in the catalog of Fig. 9.1, in terms of which more complex heuristics can be defined. This catalog is open-ended; the language implementation explicitly supports adding new

s ::= prune
prunes the node
| base_search(...)
label
| let(v, e, s)
introduce new variable v with
initial value e, then perform s
| assign(v, e)
assign e to variable v and succeed
| post(c, s)
post constraint c at every node during s
| if(c, s_1, s_2)
perform s_1 until c is false, then perform s_2
| and($[s_1, s_2, \ldots, s_n]$)
perform $s1$, on success $s2$ otherwise fail, ...
| or($[s_1, s_2, \ldots, s_n]$)
perform $s1$, on termination start $s2$, ...
| portfolio($[s_1, s_2, \ldots, s_n]$)
perform $s1$, if not exhaustive start $s2$, ...
| restart(c, s)
restart s as long as c holds

Figure 9.1: Catalog of primitive search heuristics and combinators

primitives. Primitive search heuristics consist of *basic* heuristics and *combinators*. The former define complete (albeit very basic) heuristics by themselves, while the latter alter the behavior of one or more other heuristics and combinators. The two basic search heuristics (base_search and prune) create child nodes in the search tree under the current node or prune the subtree starting from the current node, while combinators (all remaining items in Fig. 9.1) decide e.g., which of their sub-heuristics to apply or to restart search.

Note that the queuing strategy (such as depth-first traversal) is determined separately by the search engine, it is thus orthogonal to the search language.

Basic Heuristics There are two basic heuristics:

- base_search(*vars*, *var-select*, *value-select*) specifies a systematic search. If any of the variables *vars* are still not fixed at the current node, it creates child nodes according to *var-select* and *value-select* as variable- and value-selection strategies respectively. The details of the options are not explained here.

- prune cuts the search tree below the current node, resulting in a non-exhaustive search (explained below).

Note that base_search is a CP-specific primitive; other kinds of solvers provide their own search primitives. The rest of the search language is essentially solver-independent. While the solver provides few basic heuristics, the search language adds great expressive power by allowing these to be combined arbitrarily using combinators.

Combinators The expressive power of the search language relies on combinators, which combine search heuristics (which can be basic or themselves constructed using combinators) into more complex heuristics.

An example of a combinator from the literature is limited discrepancy search (LDS) (Harvey and Ginsberg 1995): lds(s,n) denotes a heuristic that performs LDS over an underlying heuristic s, which can in turn be an arbitrarily complex composition of *any* of the heuristics listed in Fig. 9.1.

The supported combinators are:

- let(v, e, s): introduces a new variable v with initial value e and visible in the search s, then continues with s.

- assign(v, e): assigns the value e to variable v and succeeds. Technically, this is not a combinator, but it is listed here as it is used in combination with let.

- if(c, s_1, s_2) evaluates condition c at every node. If c holds, then it proceeds with s_1. Otherwise, s_2 is used for the node and all its children.

- and($[s_1, \ldots, s_n]$): and-sequential composition runs s_1. At every success leaf of s_1, it runs and($[s_2, \ldots, s_n]$).
- or($[s_1, \ldots, s_n]$): or-sequential composition runs s_1. Upon fully exploring the tree of s_1, search is restarted with or($[s_2, \ldots, s_n]$) regardless of failure or success of s_1.
- portfolio($[s_1, \ldots, s_n]$), in contrast, also runs s_1 in full, but only if s_1 was not exhaustive, does it restart with portfolio($[s_2, \ldots, s_n]$) (see further details on the meaning of *exhaustiveness* in the next paragraph).
- restart(c, s): repeatedly runs s in full. If s was not exhaustive, it is restarted, until condition c no longer holds.
- post(c, s): provides access to the underlying constraint solver, posting a constraint c at every node during s. If s is omitted, it posts the constraint and immediately succeeds.

The attentive reader may have noticed that lds(s,n) is actually not listed among the primitive combinators. Indeed, Sect. 9.3.2 shows next that it is a composition of primitive combinators. Moreover, the depth-first traversal that is commonly associated with lds is entirely orthogonal to the search language.

Exhaustiveness When a search has fully explored the search (sub)tree, without purposefully skipping parts using the prune primitive, it is said to be *exhaustive*. This information is used to decide whether or not to revisit the same search node, as it happens in the portfolio and restart combinators. For instance, in case of lds($s, 10$), if the search tree defined by s has been fully explored with 10 discrepancies, there is no use in restarting with higher discrepancy bounds as that would simply reexplore the same tree.

The prune primitive is the only source of non-exhaustiveness. Combinators propagate exhaustiveness in the obvious way. E.g., and($[s_1, \ldots, s_n]$) is exhaustive if all s_i are, while portfolio($[s_1, \ldots, s_n]$) is exhaustive if one s_i is.

Statistics Several combinators are centered around a conditional expression c. In addition to the conventional syntax, such a condition may refer to one or more *statistics* variables. Such statistics are collected for the duration of a subsearch until the condition is met. For instance if(depth $< 10, s_1, s_2$) maintains the search depth statistic during subsearch s_1. At depth 10, the if combinator switches to subsearch s_2.

There are two forms of statistics: *Local statistics* such as depth and discrepancies express properties of individual nodes. *Global statistics* such as nodes, time, failures and solutions are computed for entire search trees.

9.3.2 Composite Search Heuristics

The search combinators approach draws its expressive power from the combination of primitive heuristics using combinators. The user can create new combinators by effectively defining macros in terms of existing combinators. The following examples show how to construct complex search heuristics familiar from the literature.

Limit: The limiting combinator $\mathsf{limit}(c, s)$ performs s while c is satisfied. Then it fails:

$$\mathsf{limit}(c, s) \equiv \mathsf{if}(c, s, \mathsf{prune})$$

Search can be limited using any of the statistics defined previously, or indeed create and modify a new let variable to define limits on search.

Once: The well-known $\mathsf{once}(s)$ combinator is a special case of the limiting combinator where the number of solutions is not greater than one. This is simply achieved by maintaining and accessing the solutions statistic:

$$\mathsf{once}(s) \equiv \mathsf{limit}(\mathsf{solutions} < 1, s)$$

In contrast to prune, $\mathsf{post}(\mathit{false})$ represents an *exhaustive* search without solutions. This is exploited in the exhaustive variant of once:

$$\mathsf{exh\_once}(s) \equiv \mathsf{if}(\mathsf{solutions} < 1, s, \mathsf{post}(\mathit{false}))$$

Branch-and-bound: A slightly more advanced example is the branch-and-bound optimization strategy:

$$\mathsf{bab}(\mathit{obj}, s) \equiv \mathsf{let}(\mathit{best}, \infty, \mathsf{post}(\mathit{obj} < \mathit{best}, \mathsf{and}([s, \mathsf{assign}(\mathit{best}, \mathit{obj})])))$$

which introduces a variable *best* that initially takes value ∞ (for minimization). In every node, it posts a constraint to bound the objective variable by *best*. Whenever a new solution is found, the bound is updated accordingly.

Restarting branch-and-bound: This is a twist on regular branch-and-bound that restarts whenever a solution is found.

$$\mathsf{restart\_bab}(\mathit{obj}, s) \equiv \mathsf{let}(\mathit{best}, \infty, \mathsf{restart}(\mathit{true}, \mathsf{and}([\mathsf{post}(\mathit{obj} < \mathit{best}), \mathsf{once}(s), \mathsf{assign}(\mathit{best}, \mathit{obj})]))))$$

For: The for loop construct ($v \in [l, u]$) can be defined as:

$$\mathsf{for}(v, l, u, s) \equiv \mathsf{let}(v, l, \mathsf{restart}(v \leqslant u, \mathsf{portfolio}([s, \mathsf{and}([\mathsf{assign}(v, v+1), \mathsf{prune}])]))))$$

It simply runs the search s times, which of course is only sensible if s makes use of side effects or the loop variable v. Note that assign succeeds, so prune needs to be

called afterwards in order to propagate the non-exhaustiveness of s to the restart combinator.

Limited discrepancy search with an upper limit of l discrepancies for an underlying search s.

$$\mathsf{lds}(s, l) \equiv \mathsf{for}(n,\ 0,\ l,\ \mathsf{limit}(\mathsf{discrepancies} \leqslant n, s))$$

The for construct iterates the maximum number of discrepancies n from 0 to l, while limit executes s as long as the number of discrepancies is smaller than n. The search makes use of the discrepancies statistic that is maintained by the search infrastructure. The original LDS visits the nodes in a specific order. The search described here visits the same nodes in the same order of discrepancies, but possibly in a different individual order – as this is determined by the global queuing strategy.

The following is a combination of branch-and-bound and limited discrepancy search for solving job shop scheduling problems, as described in (Harvey and Ginsberg 1995). The heuristic searches the Boolean variables *prec*, which determine the order of all pairs of tasks on the same machine. As the order completely determines the schedule, the start times are fixed using exh_once.

$$\mathsf{bab}(\mathit{makespan},\ \mathsf{lds}(\mathsf{and}([\mathsf{base\_search}(\mathit{prec}, \dots),\ \mathsf{exh\_once}(\mathsf{base\_search}(\mathit{start}, \dots))])), \infty)$$

Fully expanded, this heuristic consists of 17 combinators and is 11 combinators deep.

Iterative deepening (Korf 1985) for an underlying search s is a particular instance of the more general pattern of restarting with an updated bound.

$$\mathsf{id}(s) \equiv \mathsf{ir}(\mathsf{depth}, 0, +, 1, \infty, s)$$
$$\mathsf{ir}(p, l, \oplus, i, u, s) \equiv \mathsf{let}(n, l,\ \mathsf{restart}(n \leqslant u, \mathsf{and}([\mathsf{assign}(n, n \oplus i),\ \mathsf{limit}(p \leqslant n, s)]))))$$

With let, bound n is initialized to l. Search s is pruned when statistic p exceeds n, but iteratively restarted by restart with n updated to $n \oplus i$. The repetition stops when n exceeds u or when s has been fully explored. The bound increases geometrically, if $*$ is supplied for $\oplus$, as in the restart_flip heuristic:

$$\mathsf{restart\_flip}(p, l, i, u, s_1, s_2) \equiv \mathsf{let}(\mathit{flip}, 1, \mathsf{ir}(p, l, *, i, u, \mathsf{and}([\mathsf{assign}(\mathit{flip}, 1 - \mathit{flip}),\ \mathsf{if}(\mathit{flip} = 1, s_1, s_2)])))$$

This alternates between two search heuristics s_1 and s_2. Using this as its default strategy in the *free search* category, the lazy clause generation solver *Chuffed* scored most points in the 2010 MiniZinc Challenge.[1]

[1] http://www.g12.csse.unimelb.edu.au/minizinc/challenge2010/

Hot start: First perform search heuristic s_1 while condition c holds to initialize global parameters for a second search s_2. This heuristic is for example used for initialization of the widely applied *Impact* heuristic (Refalo 2004).

$$\mathsf{hotstart}(c, s_1, s_2) \equiv \mathsf{portfolio}([\mathsf{limit}(c, s_1), s_2])$$

Radiotherapy treatment planning: The following search heuristic can be used to solve radiotherapy treatment planning problems (Baatar, Boland, Brand, and Stuckey 2011). The heuristic minimizes a variable k using branch-and-bound (bab), first searching the variables N, and then verifying the solution by partitioning the problem along the row_i variables for each row i one at a time (expressed as a MiniZinc array comprehension). Failure on one row must be caused by the search on the variables in N, and consequently search never backtracks into other rows.

$$\begin{aligned}&\mathsf{bab}(k,\ \mathsf{and}([\mathsf{base\_search}(N, \dots)]\texttt{++}\\ &\qquad\qquad [\mathsf{exh\_once}(\mathsf{base\_search}(row_i, \dots)) \mid i \text{ in } 1..n]))\end{aligned}$$

Dichotomic Search (Sellmann and Kadioglu 2008) solves an optimization problem by repeatedly partitioning the interval in which the possible optimal solution can lie. It can be implemented by restarting as long the lower bound has not met the upper bound (line 2), computing the middle (line 3), and then using an or combinator to try the lower half (line 5). If it succeeds, $obj - 1$ is the new upper bound, otherwise, the lower bound is increased (line 6).

$$\begin{aligned}&\mathsf{dicho}(s, obj, lb, ub) \equiv \mathsf{let}(l, lb, \mathsf{let}(u, ub, \mathsf{let}(h, 0,\\ &\quad \mathsf{restart}(l < u,\\ &\quad\quad \mathsf{let}(h, l + \lceil (u - l)/2 \rceil,\\ &\quad\quad \mathsf{once}(\mathsf{or}([\\ &\quad\quad\quad \mathsf{and}([\mathsf{post}(l \leqslant obj \leqslant h), s, \mathsf{assign}(u, obj - 1)]),\\ &\quad\quad\quad \mathsf{and}([\mathsf{assign}(l, h + 1), \mathsf{prune}])])))\\ &\quad))))\end{aligned}$$

9.4 Semantics

At the lowest level, search is performed by repeatedly popping search nodes from a queue, processing them, and possibly pushing new nodes to the queue. We use a stack-based terminology here, but the underlying queue may be different.

In fact, the queue type determines the basic node ordering. Using a stack results in a DFS-based search, while a FIFO makes the search BFS-based. As explained before, this is orthogonal to the rest of the search specification.

Search Combinators can be regarded as a successor to MCP's Search Transformers (Section 4.2), which required variable labeling to be described inside the

constraint model, and search to be independent of the problem's variables. Search Combinators improves upon this by allowing different base searches for different variables to be combined into a single search heuristic. This is for example necessary in the earlier radiotherapy example.

The combinator stack For each node, and at each point during the processing, a given combination of combinators will be active. Although the AST representing the search specification is a tree, with each subtree another search specification, processing happens top-down, and only a single path through the combinator tree is active at a given time. The list of combinators along this path is called the combinator stack.

Figure 9.2 shows the combinator stack for the earlier branch-and-bound example.

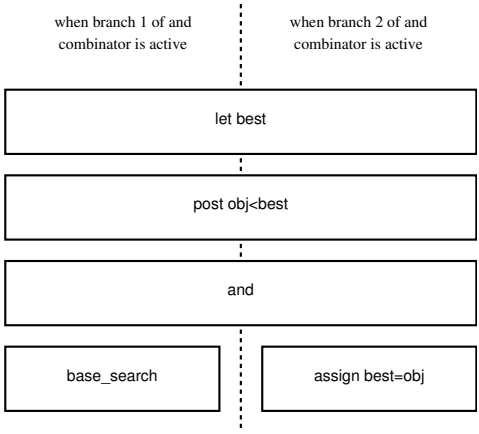

Figure 9.2: Branch-and-bound combinator stack

Node processing loop During a main processing loop, nodes are activated one by one. Doing so involves going through a series of hooks. There are several ones, each corresponding to a separate stage of processing a node. Initially, a first node is created that represents the top of the search tree. Then, the *body* hook of the top combinator is invoked on it. Typically, this will pass processing to one or more of its subcombinators. Eventually, the *body* hook of one of the base searches is reached, which restarts processing the chain, but now with the *add* hook. Similarly, *add* will eventually call *try*, and *try* will call *result*.

The purpose of these hooks is to allow combinators to modify the behavior of generated code, by letting them insert or modify statements at different stages of

processing.

Hooks The different hooks, each corresponding to one stage of processing are:

body In this stage, a check is done to verify whether it should be processed at all. Each combinator is allowed to introduce checks, and optionally stop the processing, by diverting the call to *fail*

add When this stage is reached, the node will definitely be processed. This hook is mainly used to add additional constraints to the problem.

try During this stage, checks are done to see whether branching on the node is required, and either branch, fail, or call result. This branching will cause new nodes to be added to a queue, which will again be processed, starting with *body*.

result This stage is reached when a solution has been found.

There are two additional hooks:

init Initialization of a new node.

fail Called whenever processing a node stopped.

The traversal of the hooks is depicted in Figure 9.3.

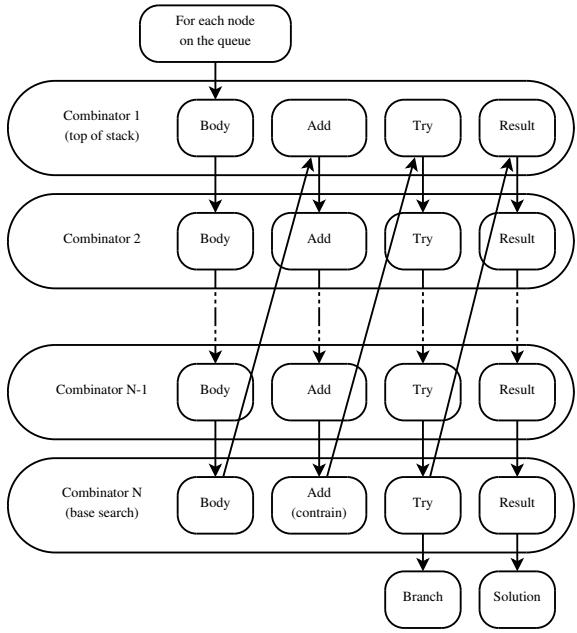

Figure 9.3: Node processing protocol

Chapter 10

Code generation for search

Analogous to Chapter 6 — where an implementation for code generation of constraints was discussed — this chapter deals with code generation of search specifications, specific challenges, and how to exploit abstractions provided by Haskell to deal with them.

10.1 Overview

The search descriptions — in the form of MiniZinc annotations, as described in the previous chapter — are processed by a parser, that maps base searches to values, and combinators to functions over them. This way, a data structure is built up that represents the search heuristic. This data structure is then processed by a code generator to produce a C++ AST. Finally, this AST is converted to actual source code by a pretty printer. Both the initial parsing phase and pretty printer are trivial and not discussed here.

10.1.1 C++ Abstract Syntax Tree

The C++ AST used to represent the generated code is defined by a Haskell ADT:

$$
\begin{aligned}
\textbf{data}\ \mathit{C{+}{+}} = {} & \mathit{Nop} && \mid \mathit{Expr} := \mathit{Expr} \\
\mid {} & \mathit{IfThenElse}\ \mathit{Expr}\ \mathit{Stmt}\ \mathit{Stmt} && \mid \mathit{Stmt}\,;\mathit{Stmt} \\
\mid {} & \mathit{Call}\ \mathit{String}\ [\mathit{Expr}] && \mid \mathit{While}\ \mathit{Expr}\ \mathit{Stmt} \\
\mid {} & \ldots
\end{aligned}
$$

A number of convenient abbreviations facilitate building this AST, e.g.,

$$
\begin{aligned}
(\tfrac{\circ}{;}) &= \mathit{liftM} \quad \circ (;) \\
\mathit{if}' &= \mathit{liftM2} \circ \mathit{IfThenElse}
\end{aligned}
$$

A no-op statement represents the absence of any code. When transforming to C++ code, these are mapped to nothing. The other constructors correspond to:

:= An assignment (= in C++)

IfThenElse An if-then-else block

; A sequence of two other (blocks of) statements

Call A function call

While A while loop

The ADT uses arguments of type *Expr*. These represent (untyped) C++ expressions, and include normal arithmetic, boolean operators, bitwise operators, function calls, ...

Similar to the expression type from Section 5.3.1, this data structure allows some low-level optimizations through application of rewrite rules.

10.2 The Code Generator

To convert the search specification to an AST, an intermediate data structure is built up that represents the entire search specification — after "resolving" the crosscutting aspects. An ADT *Gen m* is used, with fields that correspond to the hooks[1].

As will be explained later, some combinators need to keep an own modifiable state during code generation, so hooks must support side effects; hence *Gen* is parametrized in a monad *m*.

$$\begin{aligned}\textbf{data}\ \mathit{Gen}\ m = \mathit{Gen}\ \{\ & \mathit{init}_G \ :: m\ \mathit{Stmt}, \mathit{body}_G :: m\ \mathit{Stmt}\\ ,\ & \mathit{add}_G \ :: m\ \mathit{Stmt}, \mathit{try}_G \ :: m\ \mathit{Stmt}\\ ,\ & \mathit{result}_G :: m\ \mathit{Stmt}, \mathit{fail}_G \ :: m\ \mathit{Stmt}\\ ,\ & \mathit{height} \ :: \mathit{Int}\ \}\end{aligned}$$

When this *Gen m* structure is built, it is converted to C++ using this template:

$$\begin{aligned}&\mathit{gen} :: \mathit{Monad}\ m \Rightarrow \mathit{Gen}\ m \to m\ \mathit{Stmt}\\ &\mathit{gen}\ g = \textbf{do}\ \mathit{init} \leftarrow \mathit{init}_G\ g\\ &\qquad\qquad \mathit{try} \ \leftarrow \mathit{try}_G\ g\\ &\qquad\qquad \mathit{body} \leftarrow \mathit{body}_G\ g\\ &\qquad\qquad \mathit{return}\ \$\ \ \mathit{declarations}\\ &\qquad\qquad\qquad\qquad ; \mathit{init}\\ &\qquad\qquad\qquad\qquad ; \mathit{try}\\ &\qquad\qquad\qquad\qquad ; \mathit{While}\ \mathit{queueNotEmpty}\ \mathit{body}\end{aligned}$$

[1] See Section 10.2.1 for why we partition the code generation into these hooks

After emitting a number of variable declarations, the template creates the root node in the search tree through $init_G$, and try_G initializes a queue with child nodes of the root. Then, in the main part of the algorithm, nodes in the queue are processed one at a time using the $body_G$ hook.

10.2.1 Code Generation Mixins

Instead of writing a monolithic code generator for every different search heuristic, new heuristics are modularly composed from one or more components, each of which corresponds to a constructor in the high-level DSL. Our code generator components are implemented as (functional) mixins (Bracha and Cook 1990), where the result is a function from `Eval m` to `Eval m`, which gets called with its own resulting strategy as argument. The function argument in these mixins is comparable to the *this* object in object-oriented paradigms.

The base heuristics are considered self-contained, while combinators with a single other search heuristic as argument are considered *advice* components that extend or modify another component (Oliveira, Schrijvers, and Cook 2010). An alternative analogy for mixins, that includes multi-argument combinators, is that of *inheritance*, where we distinguish self-contained "base classes" and "class deltas". The application of a class delta Δ to a number of classes $\bar{C}$ yields a subclass $\Delta(\bar{C})$; this subclass is said to inherit from $\bar{C}$. When $\bar{C}$ consists of more than one class, we speak of *multiple inheritance*.

$$\begin{aligned}&\textbf{type } \mathit{Mixin}\ a = a \to a\\ &\textbf{type } \mathit{MGen}\ m = \mathit{Mixin}\ (\mathit{Gen}\ m)\end{aligned}$$

Base Component Base searches are implemented as $Gen\ m \to Gen\ m$ functions (shortened using a type alias to $MGen\ m$ here), with fix-point semantics. Using lazy evaluation, we can pass the fully combined search as an argument back to itself. Through this mechanism, we can make the base search's hooks call other hooks back at the top of the chain, as shown in the protocol overview shown in Figure 9.3.

The main example of a base component is the enumeration strategy $base_M$:

$$\begin{aligned}&\mathit{base}_M :: \mathit{Monad}\ m \Rightarrow \mathit{MGen}\ m\\ &\mathit{base}_M\ \mathit{this} =\\ &\quad \mathit{Gen}\ \{\mathit{init}_G \quad = \mathit{return}\ \mathit{Nop}\\ &\qquad , \mathit{body}_G \quad = \mathit{add}_G\ \mathit{this}\\ &\qquad , \mathit{add}_G \quad = \mathit{constrain} \fatsemi \mathit{try}_G\ \mathit{this}\\ &\qquad , \mathit{try}_G \quad = \textbf{let}\ \mathit{ret} \quad = \mathit{result}_G\ \mathit{this}\\ &\qquad\qquad\qquad\quad \mathit{succ} = \mathit{if}'\ \mathit{isSolved}\ \mathit{ret}\ \mathit{doBranch}\\ &\qquad\qquad\qquad\quad \textbf{in}\ \mathit{if}'\ \mathit{isFailed}\ (\mathit{fail}_G\ \mathit{this})\ \mathit{succ}\end{aligned}$$

$$
\begin{aligned}
&, result_G = return\ Nop \\
&, fail_G \ \ = return\ Nop \\
&, height \ \ = 0\}
\end{aligned}
$$

The above code omits details related to posting constraints (*constrain*), checking the solver status (*isSolved* or *isFailed*) and branching (*doBranch*). The details of these operations depend on the particular constraint solver involved (e.g. finite domain, linear programming, ...); here the focus lies on the search heuristics, which are orthogonal to those details.

The base component is parametrized by *this*, the overall search heuristic. This way, the $base_M$ search can make the final call to $body_G$ redirect to an add_G on the top of the combinator-stack again, restarting the processing top-down, but this time using add_G instead of $body_G$. A similar construct is used for called try_G and $result_G$.

The simplest form of a search heuristic is obtained by applying the fix-point combinator to a base component:

$$
\begin{aligned}
&fix :: Mixin\ a \to a \\
&fix\ m = m\ (fix\ m) \\
&search_1 :: Gen\ Identity \\
&search_1 = fix\ base_M
\end{aligned}
$$

Advice Component The mixin mechanism allows us to plug in additional advice components before applying the fix-point combinator. This way we can modify the base component's behavior.

Consider a simple example of an advice combinator that prints solutions:

$$
\begin{aligned}
&print_M :: Monad\ m \Rightarrow MGen\ m \\
&print_M\ super = super\ \{result_G = printSolution\ ;\ result_G\ super \\
&\qquad\qquad\qquad\qquad , height \ \ = 1 + height\ super\}
\end{aligned}
$$

where *printSolution* consists of the necessary solver-specific code to access and print the solution. A code generator is obtained through mixin composition, simply using $(\circ)$:

$$
\begin{aligned}
&search_2 :: Gen\ Identity \\
&search_2 = fix\ (print_M \circ base_M)
\end{aligned}
$$

10.2.2 Monadic Components

In the components we have seen so far, the monad type parameter m has not been used. It does become essential when we turn to more complex components such as the binary conjunction $\mathsf{and}([g_1, g_2])$.

The code presented at the end of this section shows a simplified *and* combinator, for two *Gen m* structures with the same type m. It does require m to be an instance of $Reader_M\ Side$, to store the current stage at code-generation runtime. While some hooks simply dispatch to the corresponding hook of the currently active stage, $body_G$ and $result_G$ are more elaborate.

First of all, we also need to store the stage number at program runtime. This is known at the time when the node is created, but needs to be restored into the monadic state when activating it. We assume the functions *store* and *retrieve* give access to a runtime state for each node, indexed with a field name and the height of the combinator involved.

When the $result_G$ hook is called — implying a solution for a sub-branch was found — there are two options. Either the g_1 was active, in which case both the runtime state and the monadic state are updated to In_2, and $init_G$ and try_G for g_2 are executed, which will possibly cause the node to be added to the queue, if branching is required. When this new node is activated itself, its $body_G$ hook will be called, retrieving the branch information from the runtime state, and dispatching dynamically to g_2. When a solution is reached after switching to g_2, $result_G$ will finally call g_2's $result_G$ to report the full solution.

```
data Branch = In1 | In2
type Mixin2 a = a → a → a
and_M :: Reader_M Branch m ⇒ Mixin2 (Gen m)
and_M g1 g2 = Gen { init_G    = store myHeight "pos" In1 ⨟ init_G g1
                  , add_G     = dispatch add_G
                  , try_G     = dispatch try_G
                  , fail_G    = dispatch fail_G
                  , body_G    = myBody
                  , result_G  = myResult
                  , height    = myHeight }
  where parent     = ask >>= λx → case x of
                                    In1 → return g1
                                    In2 → return g2
        dispatch f = parent >>= f
        myHeight   = 1 + max (height g1) (height g2)
        myBody     = let pos = retrieve myHeight "pos"
                         br1 = local (const In1) (body_G g1)
                         br2 = local (const In2) (body_G g2)
                     in if' (pos =:= In1) br1 br2
        myResult   = do num ← ask
                        case num of
                          In1 → local (const In2) $
                                  store myHeight "pos" In2
                                ⨟ liftM2 (;) (init_G g2) (try_G g2)
```

$$In_2 \to result_G\ g_2$$

10.2.3 Effect Encapsulation

So far we have parametrized *MGen* with m, a monad type parameter. This parameter will have to be assembled appropriately from monad transformers to satisfy the need of every mixin component in the code generator. Doing this manually can be quite cumbersome. Especially for a large number of mixin components with multiple instances of, e.g., $Reader_T$ this becomes impractical. To simplify the process, we turn to a technique proposed in (Schrijvers and Oliveira 2010a) to encapsulate the monad transformers inside the components.

$$\begin{aligned}&\textbf{data}\ Search = \forall t_2.Trans_M\ t_2 \Rightarrow\\ &\quad Search\ \{\ mgen :: \forall m\ t_1.(Monad\ m, Trans_M\ t_1) \Rightarrow MGen\ ((t_1 \rhd t_2)\ m)\\ &\qquad\quad ,\ run\quad :: \forall m\ x.\ Monad\ m \Rightarrow t_2\ m\ x \to m\ x\ \}\end{aligned}$$

To that end we now represents the search data structure using the *Search* type, which packages the components behavior *MGen* with its side effect t_2. The monad transformer t_2 is existentially quantified to remain hidden; we can eliminate it from a monad stack with the *run* field. The hooks of the component are available through the *mgen* field, which specifies them for an arbitrary monad stack in which t_2 is surrounded by more effects t_1 above and m below. Here $t_1 \rhd t_2$ indicates that the focus rests on t_2 (away from t_1) for resolving overloaded monadic primitives such as *get* and *put*, for which multiple implementations may be available in the monad stack. We refer to (Schrijvers and Oliveira 2010b; Schrijvers and Oliveira 2010a) for details of this focusing mechanism, known as the *monad zipper*.

An auxiliary function promotes a non-effectful *MGen m* to *MSearch*:

$$\begin{aligned}&\textbf{type}\ MSearch = Mixin\ Search\\ &mkSearch :: (\forall m.Monad\ m \Rightarrow MGen\ m) \to MSearch\\ &mkSearch\ f\ super =\\ &\quad \textbf{case}\ super\ \textbf{of}\\ &\qquad Search\ \{\ mgen = mgen, run = run\ \} \to Search\ \{\ mgen = f \circ mgen\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ,\ run\quad = run\ \}\end{aligned}$$

which we can apply for instance to $base_M$ and $print_M$.

$$\begin{aligned}&base_S, print_S :: MSearch\\ &base_S\ = mkSearch\ base_M\\ &print_S = mkSearch\ print_M\end{aligned}$$

Similarly, we define $mkSearch_2$ for lifting binary combinators like and_M. It takes a combinator for two *Gen m*'s, as well as a run function for additional monad transformers the combinator may require, and lifts it to *MSearch2*.

$$
\begin{aligned}
&\textbf{type } \mathit{MSearch}_2 = \mathit{Mixin}_2\ \mathit{Search}\\
&\mathit{and}_S :: \mathit{MSearch}_2\\
&\mathit{and}_S = \mathit{mkSearch}_2\ \mathit{and}_M\ (\mathit{flip}\ \mathit{runReader}_T\ \mathit{In}_1)\\
&\mathit{mkSearch}_2 :: \mathit{Trans}_M\ t_2\\
&\qquad\qquad \Rightarrow (\forall m\ t_1.(\mathit{Monad}\ m, \mathit{Trans}_M\ t_1) \Rightarrow \mathit{Mixin}_2\ (\mathit{Gen}\ ((t_1 \triangleright t_2)\ m)))\\
&\qquad\qquad \rightarrow (\forall m\ x.\mathit{Monad}\ m \Rightarrow t_2\ m\ x \rightarrow m\ x)\\
&\qquad\qquad \rightarrow \mathit{MSearch}_2
\end{aligned}
$$

Finally we produce C++ code from a *Search* component with *generate*:

$$
\begin{aligned}
&\mathit{generate} :: \mathit{Search} \rightarrow \mathit{Stmt}\\
&\mathit{generate}\ s = \textbf{case}\ s\ \textbf{of}\\
&\qquad\qquad \mathit{Search}\ \{\mathit{mgen} = \mathit{mgen}, \mathit{run} = \mathit{run}\} \rightarrow\\
&\qquad\qquad\quad \mathit{runIdentity}\ \$\ \mathit{run}\ \$\ \mathit{runIdentity}_T\ \$\ \mathit{runZ}\ \$\ \mathit{gen}\ \$\ \mathit{fix}\ \$\ \mathit{mgen}
\end{aligned}
$$

This code first applies the fix-point computation, passing the result back into itself, as explained earlier. After that, *gen* is called to get the real code-generating monad action. It extracts the knot-tied body_G hook, *runZ* eliminates $\triangleright$ from $(t_1 \triangleright t_2)\ m$, yielding $t_1\ (t_2\ m)$. Then $\mathit{runIdentity}_T$ eliminates t_1 (instantiating it to be $\mathit{Identity}_T$), *run* eliminates t_2, and *runIdentity* finally eliminates m (instantiating it to be *Identity*) to yield a *Stmt*.

10.3 Memoization and Inlining

Experimental evaluation indicates that several component hooks in a complex search heuristic are called frequently, as for example the fail_G hook can be called from many different places. This is a problem for

- the code generation, which needs to generate the corresponding code over and over again
- the generated program which contains much redundant code.

Both significantly impact the compilation time (in Haskell and in C++); in addition, an overly large binary executable may aversely affect the cache and ultimately the running time.

10.3.1 Basic Memoization

A well-known approach that avoids the first problem, repeatedly computing the same result, is *memoization*. Fortunately, Brown and Cook (Brown and Cook 2009) have shown that memoization can be added as a monadic mixin component in a straightforward way.

Memoization is a side effect for which we define a custom monad transformer:

```
newtype 𝕄_T m a = 𝕄_T (State_T Table m a)
  deriving (Trans_M)
run𝕄_T :: Monad m ⇒ 𝕄_T m a → m (a, Table)
run𝕄_T (𝕄_T m) = runState_T m initMemoState
```

which is essentially a state transformer that maintains a table from *Key*s to *Stmt*s. For now we use *String*s as *Key*s.

```
newtype Key   = String
newtype Table = Map Key Stmt
initMemoState = empty
```

We capture the two essential operations of $\mathbb{M}_T$ in a type class, which allows us to lift the operations through other monad transformers.

```
class Monad m ⇒ 𝕄_M m where
  get𝕄 :: String → m (Maybe Stmt)
  put𝕄 :: String → Stmt → m ()
instance Monad m ⇒ 𝕄_M (𝕄_T m) where ...
instance (𝕄_M m, Trans_M t) ⇒ 𝕄_M (t m) where ...
```

These operations are used in an auxiliary mixin function:

```
memo :: 𝕄_M m ⇒ String → Mixin (m Stmt)
memo s m = do stm ← get𝕄 s
              case stm of
                Nothing → do code ← m
                             put𝕄 s code
                             return code
                Just code → return code
```

which is used by the advice component:

```
memo_M :: 𝕄_M m ⇒ MGen m
memo_M super = super {init_G   = memo "init"   (init_G   super)
                     , body_G  = memo "body"   (body_G  super)
                     , add_G   = memo "add"    (add_G   super)
                     , try_G   = memo "try"    (try_G   super)
                     , result_G = memo "result" (result_G super)
                     , fail_G  = memo "fail"   (fail_G  super)}
```

which allows us to define, e.g., a memoized variant of $print_S$.

$$print_S = mkSearch\ (memo_M \circ print_M)$$

Note that in order to lift $memo_M$ to a *Search* structure, *Search* must be updated with a $\mathbb{M}_M\ m$ constraint, and *generate* must be updated to incorporate $run\mathbb{M}_T$ in its evaluation chain.

$$\begin{aligned}
&\textbf{data}\ Search = \forall t_2.Trans_M\ t_2 \Rightarrow \\
&\quad Search\ \{\, mgen :: \forall m\ t_1.(\mathbb{M}_M\ m, Trans_M\ t_1) \Rightarrow MGen\ ((t_1 \triangleright t_2)\ m) \\
&\qquad\quad , run \quad :: \forall m\ x.\ \mathbb{M}_M\ m \Rightarrow t_2\ m\ x \to m\ x \,\} \\
&generate\ s = \\
&\quad \textbf{case}\ s\ \textbf{of} \\
&\qquad Search\ \{\, mgen = mgen, run = run \,\} \to \\
&\qquad\quad runIdentity\ \$\ run\mathbb{M}_T\ \$\ run\ \$\ runIdentity_T\ \$\ runZ\ \$\ gen\ \$\ fix\ mgen
\end{aligned}$$

10.3.2 Monadic Memoization

Unfortunately, the implementation is not quite this simple. The behavior of combinator hooks may depend on internal updateable state, like and_M from Section 10.2.2 kept a *Branch* value as state. The above memoization does not take this state dependency into account.

In order to solve this issue, we must expose the components' state to the memoizer. This is done in two steps. First, $\mathbb{M}_T$ keeps a *context* in addition to the memoization table, and provides access to it through the $\mathbb{M}_M$ type class. Second — for the specific case of a $Reader_T\ s$ with s an instance of *Show* — an alternative implementation ($\mathbb{M}Reader_T$) which updates the context in the $\mathbb{M}_T$ layer below it, is provided. Typically, the used states are simple in structure.

To implement this, the *Table* type is extended:

$$\begin{aligned}
&\textbf{type}\ MemoContext = Map\ Int\ String \\
&\textbf{type}\ Key \qquad\qquad = (MemoContext, String) \\
&\textbf{data}\ Table = Table\ \{\, context :: MemoContext \\
&\qquad\qquad\qquad\qquad , memoMap :: Map\ Key\ Stmt \,\} \\
&initMemoState = Table\ \{\, context \quad = empty \\
&\qquad\qquad\qquad\qquad , memoMap = empty \,\}
\end{aligned}$$

MemoContext is represented as a map from integers to strings. The integers are identifiers assigned to the monad transformer layers that have context, and the strings are serialized versions of the contextual data inside those layers (using *show*).

The $\mathbb{M}_M$ type class is extended to support modifying the context information, using *setCtx* and *clearCtx*.

```
class Monad m ⇒ 𝕄_M m where
  ...
  setCtx :: Int → String → m ()
  clearCtx :: Int → m ()
```

Finally, $\mathbb{M}Reader_T$ is introduced. It will contain a wrapped double $Reader_T$-transformed monad. The state will be stored in the first, while the second is used to give access to the identifier of the layer.

```
newtype 𝕄Reader_T s m a =
  𝕄Reader_T {r𝕄Reader_T :: Reader_T Int (Reader_T s m) a}
```

For convenience, $\mathbb{M}Reader_T$ is made an instance of $Reader_M$, so switching from $Reader_T$ to $\mathbb{M}Reader_T$ does not require any changes to the code interacting with it.

When running a $\mathbb{M}Reader_T$ transformer, the enclosing *Gen*'s *height* parameter is passed to $runReader_T$, using that as identifier for the layer. The runtime state itself is stored inside the wrapped $Reader_T$ layer, while a serialized representation (using *show*) is stored in the context of the underlying $\mathbb{M}_T$. Note that *show* implementations are supposed to turn a value into equivalent Haskell source code for reconstructing the value — this is far from the most efficient solution, but it does produce canonical descriptions for all values, and default implementations are provided by the system for almost all useful data types. There are alternatives, such as using an *Ord*-providing *Dynamic*-like type, but those are harder to implement and there is little to be gained, as is shown in the evaluation (Section 10.4).

```
instance (Show s, 𝕄_M m) ⇒ Reader_M s (𝕄Reader_T s m) where
  ask = 𝕄Reader_T $ lift ask
  local s m = 𝕄Reader_T $ do n ← ask
                             old ← lift ask
                             let new = s old
                             putCtx n $ show new
                             let im = run𝕄Reader_T m
                             r ← mapReaderT (local $ const new) im
                             putCtx n $ show old
                             return r
run𝕄Reader_T :: (𝕄_M m, Show s) ⇒ s → Int → 𝕄Reader_T s m a → m a
run𝕄Reader_T s height m =
  do let action = runReader_T (r𝕄Reader_T m) height
     putCtx height (show s)
     result ← runReader_T action s
     clearCtx height
     return result
```

10.3.3 Back-end Sharing

So far we have only solved the first performance problem, repeated generation of code. Memoization avoids the repeated execution of hooks by storing and reusing the same C++ code fragment. However, the second performance problem, repeated output of the same C++ code, remains.

We preserve the sharing obtained through memoization in the back-end, by depositing the memoized code fragment in a C++ function that is called from multiple sites. Conceptually, this means that a memoized hook returns a function call (rather than a potentially big code fragment), and produces a function definition as a side effect.[2]

```
memo₂ :: 𝕄_M m ⇒ String → Mixin (m Stmt)
memo₂ s m = do code ← memo s m
               let name = getFnName code
               return (Call name [])
getFnName :: Stmt → String
```

The following *generate* function produces both the main search code and the auxiliary functions for the memoized hooks. By introducing $run\mathbb{M}_T$ in the chain of evaluation functions, the types change, and the result will be of type (*Stmt*, *Table*), since that is returned by $run\mathbb{M}_T$.

```
data FunDef = FunDef String Stmt
toFunDef :: Stmt → FunDef
toFunDef stm = FunDef (getFnName stm) stm
generate :: Search → (Stmt, [FunDef])
generate s =
  case s of
    Search {mgen = mgen, run = run} →
      let eval          = fix mgen
          codeM         = gen eval
          memoM         = run ∘ runIdentity_T ∘ runZ $ codeM
          (code, state) = runIdentity $ run𝕄_T memoM
          in (code, map toFunDef ∘ elems $ memoMap state)
```

The result of extracting common pieces of code into separate functions, is shown schematically in figure 10.1.

Note that only code generated by the same hook of the same component is shared in a function, not code of distinct hooks or distinct components. Separate from the mechanism described above, it is also possible to detect unrelated

[2]The function *getFnName* — given without implementation — derives a unique function name for a given code fragment.

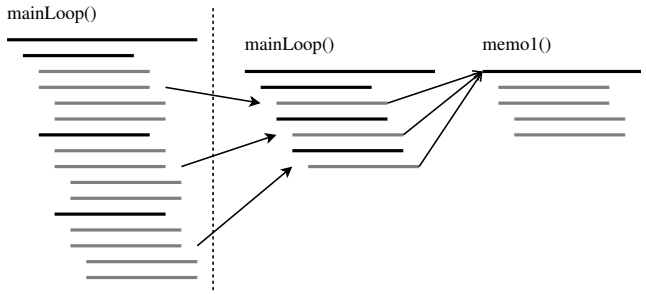

Figure 10.1: Memoization with auxiliary functions

clones by doing memoization with only the generated code itself as key (instead of function names, present variables and active states). This causes a slowdown, as the code needs to be generated for each instance before it can be recognized as identical to earlier emitted code. To a limited extent, this second memoization scheme is also used in the implementation to reduce the size of generated code — without any measurable overhead.

Finally, applying the above technique systematically results in one generated C++ function per component hook. This is not entirely satisfactory, as many memoized functions are only called once, or only contain a single line of code. One can either rely on the C++ compiler to determine when inlining is lucrative, or perform inlining on the C++ AST in an additional processing step.

10.4 Evaluation

We have omitted a number of complicating factors in our account, so as not to distract from the main issues. Without going into detail, we list the main differences with the actual implementation:

- There are more hooks, including ones called during branching, adding to the queue, deletion of nodes and switching between nodes belonging to separate strategies. Furthermore, additional hooks exist for the creation of combinator-specific data structures, both globally for the whole combinator, or locally for each node, instead of the dynamic *height*-based mechanism.
- The code generation hooks are functions that take an additional argument, the *path info*. It contains which variable names point to the local and global data structures, which variables need to be passed to generated memoized functions, and pieces of code that need to be executed when the current node needs to be stored, aborted or copied. The values in the path info are also taken into account when memoizing, complicating matters further.

- We have built into the code generators a number of optimizations. For example, if it is known that a combinator never branches, certain generated code and data structures may be omitted.

- Searches keep track of whether they complete exhaustively, or are pruned. Repeat-like combinators use exhaustiveness as an additional stop criterion.

To evaluate the usefulness of our system, benchmarks were performed (see Table 10.1)[3]. A first set includes the known problems `golfers`[4], `golomb`[5], `open stacks` and `radiation` (Baatar, Boland, Brand, and Stuckey 2011); a second set contains artificial stress tests. The different problem sizes for `golomb` use the same search code, while in `ortest` and `radiation`, different code is used.

The first three columns give the name, problem size and whether or not the memoizing version was used. Further columns show the number of generated C++ lines (col. 4), the number of invoked hooks (col. 5), the number of monad transformers active (both the effective ones (col. 6), and including $Identity_T$ and $\triangleright$ (col. 7)). Finally, the average generation (Haskell, col. 8), build (gcc, col. 9) and run time (col. 10) are listed. All these numbers are averages over many runs (of up to an hour of runtime).

For the larger problem instances, memoization reduces both generation time and build time, by reducing the number of generated lines. No reduced cache effects resulting from memoizing large generated code are observed in these examples, but performance is not affected either by the increased number of function calls. In particular for the `radiation` example, the effect of memoization is drastic. On the other hand, for small problems, memoization does not help, but the overhead is very small.

10.5 Related Work

We were inspired by the monadic mixin approach to memoization of Brown and Cook (Brown and Cook 2009). The problem of memoization of stateful monad components is not yet solved in general, but typically requires some way for exposing the implicit state, as shown in (Brown and Cook 2006) for parser combinators. In our system, this is accomplished by also memoizing the implicit state.

A different approach that results in smaller code generated from a DSL is *observable sharing* (Claessen and Sands 1999; Gill 2009). Yet, the main intent of observable sharing is quite different. Its aim is to preserve sharing at the level of Haskell in the resulting generated code, typically using *unsafePerformIO*. It does

[3] A 2.13GHz Intel(R) Core(TM)2 Duo 6400 system, with 2GiB of RAM was used. The system was running Ubuntu 10.10 64-bit, with GCC 4.4.4, Gecode 3.3.1 and Minizinc 1.3.1.

[4] Social golfer problem, CSPlib problem 10

[5] Golomb rulers, CSPlib problem 6

name	size	memo?	lines	hooks	trans.		time (s)		
					eff.	total	generate	build	run
golomb	10	no	216	70	4	14	0.00017	2.0	4.9
		yes	187	95	5	17	0.0073	2.0	4.9
	11	no							110
		yes							110
	12	no							1200
		yes							1200
open-stacks	30	no	216	70	4	14	0.00016	2.1	0.12
		yes	187	95	5	17	0.0074	2.0	0.12
golfers		no	119	29	3	8	0.00017	2.0	1.3
		yes	114	46	4	11	0.00017	2.0	1.3
radiation	15	no	11455	4153	4	76	0.57	16	210
		yes	2193	1155	5	79	0.19	4.0	230
	5	no	2530	898	4	36	0.073	4.3	0.10
		yes	933	485	5	39	0.055	2.7	0.10
bab-real		no	216	70	4	14	0.00019	2.0	17
		yes	187	95	5	17	0.0074	2.0	17
bab-restart		no	1499	1166	5	20	0.045	2.8	17
		yes	433	262	6	23	0.026	2.2	17
for+copy		no	1164	414	5	14	0.016	2.4	8.9
		yes	494	180	6	17	0.0066	2.1	8.9
once-sequence		no	2530	898	4	36	0.073	4.2	2.7
		yes	933	485	5	39	0.054	2.7	2.6
ortest	10	no	1597	849	13	48	0.11	3.2	17
		yes	1222	655	14	51	0.11	2.6	17
	20	no	4232	1869	23	88	0.82	9.7	17
		yes	3352	1465	24	91	0.79	6.7	17

Table 10.1: Benchmark results

not detect distinct calls that result in the same code, and is hard to integrate with code-generating monadic computations as appear in our setting.

Our work is directly inspired by earlier work on the Monadic Constraint Programming DSL (Schrijvers, Stuckey, and Wadler 2009; Wuille and Schrijvers 2011). In particular, we have studied how to compile high-level problem specifications in Haskell to C++ code for the Gecode library (Wuille and Schrijvers 2009b).

10.6 Conclusions

We have shown how to implement a code generator for declarative specification of a search heuristic using monadic mixins. Using this mixin-based approach, search combinators can be implemented in a modular way, and still independently modify the behavior of the generated code. Through existential types and the monad zipper, all combinators can introduce their own monad transformers to keep their own state throughout the code generation, without affecting any other

transformers.

Since the naive approach leads to certain hooks being invoked many times over, we turn to memoization to avoid code duplication. Memoization is implemented as another monadic mixin which is added transparently to existing combinators.

The system is implemented as a Haskell program that generates search code in C++ from a search specification in MiniZinc which is then further integrated in a CP solver (Gecode). Our benchmarks demonstrate the impact of memoizing the monadic mixins.

Chapter 11

Conclusion

We have shown how Domain Specific Languages with high-level specifications for constraint problems and their search heuristics can be defined and implemented in the functional programming language Haskell.

For the modeling of constraint problems themselves, we augmented the *Monadic Constraint Programming* framework with a Finite Domain-specific subsystem. The modeling interface is exposed through an FD-specific DSL embedded in Haskell, allowing reuse of its abstractions. As this DSL is independent from the actual solver used, it allows for easy experimentation. Behind this front-end language, an intermediate layer assists in translating the declarative constraint model to low-level, solver-specific constraints. To do so, it calls solver-specific backends which plug into it.

To avoid inefficiencies caused by the high-level modeling, a graph-based optimization framework was designed, which works on the entire model instead of on individual constraints. The graph representation includes higher-level constructs like *map* and *fold*, and problems with uninstantiated parameters — problem classes — are supported, requiring only a single invocation of the processing algorithm. To further prevent performance overhead during solving, the processed model can be converted to a C++ program which will perform the actual search in a later stage.

In addition to declarative modeling of the constraint problem itself, we also looked at the modeling of search heuristics. Here we built upon the *Search Combinators* framework. Again, a DSL was defined to describe search strategies. This DSL is parsed and mapped to Haskell functions which build up a mixin-based stateful code generator. The code generator is then invoked to produce a C++ program as well. To prevent inefficiencies in the generated code, we turn to automated memoization and inlining of code, which prove challenging in the context of monadic code generators.

11.1 Publications

There were several publications relating to our constraint modeling system:

- In (Wuille and Schrijvers 2009b) a first FD layer was introduced. It described a solver-agnostic FD modeling language (see Section 5.3) that still distinguished between constraints and expressions. Its only solving backend used the Overton solver. The translation system was based on the decompositional approach from Section 5.6.1. In addition, an initial Gecode code generation system was present. Reified constraints were not supported, but a specific translation scheme allowed for disjunctive models nonetheless. This scheme was later abandoned in favor of reification. A more elaborate explanation is given in (Wuille and Schrijvers 2009a). Experiments showed that FD-MCP models are significantly more compact than hand-written Gecode, without seriously affecting performance.

- Parametrization was added to the FD layer in (Wuille and Schrijvers 2011). As described in Chapter 6, this includes parameters in the front-end language, support for deferred values at compilation time, lists with parametrized lengths, and iteration constructs. Furthermore, two extra solvers were added that used Gecode for solving at runtime (see Section 5.6.2). Again, benchmarks showed that in most cases the overhead of the higher-level modeling is small. Using the fixed search or generated C++ code has even lower overhead, at a cost of less flexibility.

- The general graph-based translation system for constraint models from Chapter 7 was introduced in (Wuille and Schrijvers 2010), together with real support for reification and whole-model optimization. Through the use of Gecode-specific optimizations for counting and summing in models, the class of problems that can be solved at high efficiency was extended.

Similarly, these were publications relating to our search modeling system:

- (Schrijvers, Tack, Wuille, Samulowitz, and Stuckey 2011a) and (Schrijvers, Tack, Wuille, Samulowitz, and Stuckey 2011b) introduced the search combinators system as described in Chapter 9. Based on earlier proposals for declarative search specifications, it defines a language compatible with MiniZinc's annotation language together as well as semantics for its execution. The specifications allow a modular implementation of combinators using mixins. Examples demonstrate that many existing search heuristics can be implemented concisely using search combinators, with very low overhead compared to manual implementations.

- (Wuille, Schrijvers, Samulowitz, Tack, and Stuckey 2011) shows how to implement a code generator for search combinators using monadic mixins.

Through existential types and the monad zipper, all combinators can introduce their own monad transformers to keep their own state throughout the code generation, without affecting any other transformers. Furthermore, it is shown how to implement memoization as an additional monadic mixin that cooperates with the other mixins. Experiments show that the usage of memoization significantly improves the code generation and compilation time without affecting performance.

11.2 Future work

For future work, several interesting improvements to our system can be explored. These include, but are not limited to:

Additional data types Booleans and integers suffice for many interesting CP problems, but more types of constraint variables can be added to FD-MCP. This includes constructs that are typically supported by underlying solvers like set variables, but also compound types built on top thereof, such as tuples and maps. This possibility is explored in (De Koninck, Brand, and Stuckey 2010). One specifically interesting possibility is the addition of first-class function types to the expression language. Depending on the type and domain of the arguments these constraint function variables are applied to, they can be mapped to arrays or maps internally. This information could be extracted using the annotation system from Section 7.3.2.

Further integration of the EDSL Several new developments in Haskell allow better integration with the constraint modeling DSL. One option is providing an alternative and more polymorphic Prelude (Haskell standard library) to reuse standard functions like *not*, *fold*, *map*, ... The same can be achieved by using Template Haskell (TH), which allows a programmable preprocessing pass over the source code. Another possibility is the usage of *monad comprehensions* — a generalization of list comprehensions — to provide user-friendly construction of deferred lists.

Standard library MCP and FD-MCP both provide syntactic sugar and a number of standard utility routines for writing models. One of the strengths of our approach, using the abstractions provided by the host language Haskell, can be exploited further however, by having a standard library. Often recurring patterns, such as list processing, 2-dimensional arrays or skeletons for scheduling problems could be provided by default. As these can be implemented on top of the lower-level primitives, they can be developed independently and do not complicate the core.

Integration of constraint modeling and search Although the current FD-MCP implementation uses Search Combinators to produce the search code, the integration can be improved by allowing search to be specified within the MCP model directly. This implies extending the Haskell implementation of Search Combinators with direct solving support, as it is currently limited to a code generation approach.

Generalize code generation The memoization system presented in Chapter 10 is generally useful for code generation, but is currently only used for the C++ search code. This can be improved upon by unifying the code generation layer used by both subsystems, and incorporating the memoization logic into it. A step further is the development of a full EDSL for C++ code in Haskell that supports direct evaluation within the *IO* monad as well as memoized code generation.

Bibliography

Appel, A. W. 1998. *Modern Compiler Implementation in C*. Cambridge University Press, Chapter Instruction Selection. 7.3.2

Baatar, D., Boland, N., Brand, S., and Stuckey, P. 2011. CP and IP approaches to cancer radiotherapy delivery optimization. *Constraints*. 9.3.2, 10.4

Bracha, G. and Cook, W. R. 1990. Mixin-based inheritance. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 303–311. 5.4.3, 10.2.1

Brown, D. and Cook, W. R. 2006. Function inheritance: Monadic memoization mixins. Report, Department of Computer Sciences, University of Texas at Austin. June. 10.5

Brown, D. and Cook, W. R. 2009. Function inheritance: Monadic memoization mixins. In *Brazilian Symposium on Programming Languages (SBLP)*. 10.3.1, 10.5

Carette, J., Kiselyov, O., and Shan, C.-c. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program. 19*, 509–543. 5.3.1

Cheney, J. and Hinze, R. 2003. First-class phantom types. Tech. rep., Cornell University. 3.7.3, 4.1.4

Church, A. 1940. A formulation of the simple theory of types. *J. Symb. Log. 5,* 2, 56–68. 3.3.1

Cipriano, R., Gaspero, L. D., and Dovier, A. 2009. A hybrid solver for large neighborhood search: Mixing gecode and easylocal $^{+}$ $^{+}$. In *Hybrid Metaheuristics*, M. J. Blesa, C. Blum, L. D. Gaspero, A. Roli, M. Sampels, and A. Schaerf, Eds. Lecture Notes in Computer Science, vol. 5818. Springer, 141–155. 7.4

Claessen, K. and Sands, D. 1999. Observable sharing for functional circuit description. In *Proceedings of the 5th Asian Computing Science Conference*

on Advances in Computing Science. ASIAN '99. Springer-Verlag, London, UK, 62–73. 10.5

De Koninck, L., Brand, S., and Stuckey, P. J. 2010. Data independent type reduction for Zinc. In *16th International Conference on Principles and Practice of Constraint Programming*. Edinburgh, Scotland, Published online, no page numbers. 11.2

Duck, G. J., Stuckey, P. J., and Brand, S. 2006. ACD term rewriting. In *ICLP*, S. Etalle and M. Truszczynski, Eds. LNCS, vol. 4079. 117–131. 7.4

Fernandez, A. J., Hortala-Gonzalez, T., Saenz-Perez, F., and Del Vado-Virseda, R. 2007. Constraint functional logic programming over finite domains. *Theory Pract. Log. Program. 7,* 5, 537–582. 4

Frisch, A. M., Grum, M., Jefferson, C., Hernndez, B. M., and Miguel, I. 2005. The essence of ESSENCE: A constraint language for specifying combinatorial problems. In *In Proceedings of the 20th International Joint Conference on Artificial Intelligence*. 73–88. 1.1, 4

Frühwirth, T. 1998. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming 37,* 1–3, 95–138. 1.4

Gecode Team. 2006. Gecode: Generic constraint development environment. Available from `http://www.gecode.org`. 1.1, 2, 4, 5.6.2

Gent, I. P., Jefferson, C., and Miguel, I. 2006. Minion: A fast scalable constraint solver. In *In: Proceedings of ECAI 2006, Riva del Garda*. IOS Press, 98–102. 2

Gent, I. P. and Smith, B. M. 2000. Symmetry breaking in constraint programming. In *Proceedings of ECAI-2000*. IOS Press, 599–603. 5.3

Gill, A. 2009. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. Haskell '09. ACM, New York, NY, USA, 117–128. 10.5

Girard, J. Y. 1971. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Scandinavian Logic Symposium*, J. E. Fenstad, Ed. North-Holland, 63–92. 3.3.2

Hanus, M., Kuchen, H., and Moreno-Navarro, J. J. 1995. Curry: A truly functional logic language. 4

Harvey, W. D. and Ginsberg, M. L. 1995. Limited discrepancy search. In *IJCAI*. Morgan Kaufmann Publishers Inc., 607–613. 4.2, 9.3.1, 9.3.2

Korf, R. E. 1985. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell. 27*, 97–109. 9.3.2

Kornstaedt, L. 2001. Alice in the land of Oz – an interoperability-based implementation of a functional language on top of a relational language. In *Proceedings of the First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01), Electronic Notes in Computer Science*. Vol. 59. Elsevier Science Publishers, Firenze, Italy. 4

Liang, S., Hudak, P., and Jones, M. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '95. ACM, New York, NY, USA, 333–343. 3.6.3

Marriott, K. et al. 2008. The design of the Zinc modelling language. *Constraints 13,* 3, 229–267. 4, 7.4

Mcbride, C. and Paterson, R. 2008. Applicative programming with effects. *J. Funct. Program. 18*, 1–13. 3.6.4, 6.3.1

McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM 3*, 184–195. 3.1

Milner, R., Tofte, M., and Macqueen, D. 1997. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA. 3.6.1

Mozart Team. 2004. The mozart programming system. Available from http://www.mozart-oz.org/. 7.4

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In *Thirteenth International Conference on Principles and Practice of Constraint Programming*, C. Bessire, Ed. Lecture Notes in Computer Science, vol. 4741. Springer-Verlag, Providence, RI, USA, 529–543. 1.1, 9.1, 9.3

Oliveira, B., Schrijvers, T., and Cook, W. R. 2010. Effectiveadvice: disciplined advice with explicit effects. In *AOSD*, J.-M. Jézéquel and M. Südholt, Eds. ACM, 109–120. 10.2.1

Peyton Jones, S. et al. 2003. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming 13,* 1 (Jan), 0–255. 3.7

Pierce, B. C. 2002. *Types and programming languages*. MIT Press, Cambridge, MA, USA. 3.3

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence 9*, 268–299. 2.4

Refalo, P. 2004. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming - CP97*. Lecture Notes in Computer Science. Springer. 9.3.2

Rendl, A. 2010. Effective compilation of constraint models. Ph.D. thesis, University of St. Andrews. http://www.cs.st-andrews.ac.uk/~andrea/. 7.3.1, 7.4

Samulowitz, H., Tack, G., Fischer, J., Wallace, M., and Stuckey, P. 2010. Towards a lightweight standard search language. In *Constraint Modeling and Reformulation (ModRef'10)*, J. Pearson and T. Mancini, Eds. 9.2

Schrijvers, T. and Oliveira, B. 2010a. Modular components with monadic effects. In *Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010)*. Number UU-CS-2010-020. 264–277. 10.2.3

Schrijvers, T. and Oliveira, B. 2010b. The monad zipper. Report CW 595, Dept. of Computer Science, K.U.Leuven. 10.2.3

Schrijvers, T., Stuckey, P., and Wadler, P. 2009. Monadic constraint programming. *Journal of Functional Programming*. 1.2, 1.3, 4, 4.2, 10.5

Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., and Stuckey, P. J. 2011a. Search combinators. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*. accepted. 1.3, 11.1

Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., and Stuckey, P. J. 2011b. Search combinators. In *Proceedings of the 8th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*. Late Breaking Abstract, accepted. 11.1

Sellmann, M. and Kadioglu, S. 2008. Dichotomic search protocols for constrained optimization. In *Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*. CP '08. Springer-Verlag, Berlin, Heidelberg, 251–265. 9.3.2

Smith, J. B. 2006. *Practical OCaml*. Apress. 3.6.1

Stuckey, P. J., De La Banda, M. G., Maher, M., Slaney, J., Somogyi, Z., Wallace, M., and Walsh, T. 2005. The G12 project: Mapping solver independent models to efficient solutions. In *Proceedings of the 21st International Conference on Logic Programming, number 3668 in LNCS*. Springer-Verlag, 9–13. 7.4

Sussman, G. J. and Steele, Jr., G. L. 1998. Scheme: A interpreter for extended lambda calculus. *Higher Order Symbol. Comput. 11*, 405–439. 3

Van Hentenryck, P. 1999. *The OPL optimization programming language*. MIT Press. 4

Van Hentenryck, P. and Michel, L. 2005. *Constraint-Based Local Search*. MIT Press. 1.1, 4, 9.1

Van Weert, P., Wuille, P., Schrijvers, T., and Demoen, B. 2008. CHR for imperative host languages. In *Constraint Handling Rules (series LNCS: 5388)*, T. Schrijvers and T. Frühwirth, Eds. Springer, 161–212. 1.4

Wadler, P. 1991. Is there a use for linear logic? ACM Press, 255–273. 3.6.1

Wadler, P. 1995. Monads for functional programming. In *Advanced Functional Programming*. London, UK, 24–52. 3.6.3

Wadler, P. and Blott, S. 1989. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 60–76. 3.5

Wuille, P. and Schrijvers, T. 2008. Breaking the complexity barrier of pure functional programs with impure data structures. In *Pre-proceedings of the 20th International Symposium on Implementation and Application of Functional Languages,*. 17–32. 1.4

Wuille, P. and Schrijvers, T. 2009a. The FD-MCP framework. CW Reports, Department of Computer Science, K.U.Leuven. Aug. 11.1

Wuille, P. and Schrijvers, T. 2009b. Monadic Constraint Programming with Gecode. In *Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation*, A. M. Frisch and J. Lee, Eds. 171–185. 1.3, 6.1, 10.5, 11.1

Wuille, P. and Schrijvers, T. 2010. Expressive models for Monadic Constraint Programming. In *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation*, T. Mancin and J. K. Pearson, Eds. 1.3, 11.1

Wuille, P. and Schrijvers, T. 2011. Parameterized models for on-line and off-line use. In *WFLP 2010 Post-conference Proceedings*, J. Marino, Ed. Springer-verlag. 1.3, 10.5, 11.1

Wuille, P., Schrijvers, T., and Demoen, B. 2007. CCHR: the fastest CHR implementation, in C. In *Proceedings of the 4th Workshop on Constraint Handling Rules*. 123–137. 1.4

Wuille, P., Schrijvers, T., Samulowitz, H., Tack, G., and Stuckey, P. J. 2011. Memoizing a monadic mixin DSL. In *Proceedings of The 20th International Workshop on Functional and (Constraint) Logic Programming*. accepted. 1.3, 11.1

Index

Biography

Pieter Wuille was born on the 10th of september 1984 in Leuven, Belgium. He went to school in the Sint-Albertus College Haasrode, where he graduated in 2002. Afterwards he studied Civil Engineering at the Katholieke Universiteit Leuven, option Computer Science. His master's thesis was titled "CCHR: The fastest CHR implementation", and was supervised by Prof. dr. Bart Demoen.

In august 2007, Pieter started working as a Ph.D. student at the Department of Computer Science of the Katholieke Universiteit Leuven, in the Analysis subgroup of DTAI — Declarative Languages and Artificial Intelligence. After initally working on improving the CCHR system, and doing research on automatically selected data structures, he started working on the Monadic Constraint Programming system with Prof. dr. ir. Tom Schrijvers. In 2007, Pieter was part of the winning team of the 14th Prolog Programming Contest at ICLP'07 in Porto (Portugal).

List of Publications

Articles in international reviewed journals

Peter Van Weert, Pieter Wuille, Tom Schrijvers and Bart Demoen, *CHR for imperative host languages*, LNAI, Special Issue on Recent Advances in Constraint Handling Rules

Contributions at international conferences

Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz and Peter J. Stuckey, *Search Combinators*, CP'11

Contributions at international workshops

Pieter Wuille, Tom Schrijvers and Bart Demoen, *CCHR: the fastest CHR implementation, in C*, CHR'07

Pieter Wuille and Tom Schrijvers, *Breaking the complexity barrier of pure function programs with impure data structures*, IFL'08

Pieter Wuille and Tom Schrijvers, *Monadic Constraint Programming with Gecode*, ModRef'09

Pieter Wuille and Tom Schrijvers, *Parametrized models for on-line and off-line use*, WFLP'10

Pieter Wuille and Tom Schrijvers, *Expressive models for Monadic Constraint Programming*, ModRef'10

Pieter Wuille, Tom Schrijvers, Horst Samulowitz, Guido Tack and Peter J. Stuckey, *Memoizing a Monadic Mixin DSL*, WFLP'11

Technical reports

Pieter Wuille and Tom Schrijvers, *The FD-MCP framework*, Report CW562

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science
Declarative Languages and Artificial Intelligence
Celestijnenlaan 200A
B-3001 Heverlee

KATHOLIEKE UNIVERSITEIT LEUVEN

ASSOCIATIE K.U.LEUVEN